AIR TRAFFIC SIMULATION

Kristine Frank

Dan Erik Petersen

Mark Hoffmann

Christian Krog Madsen

10. May 2000

49156 Informatics Project

Department of Information Technology

Technical University of Denmark

Contents

1 Introduction 3

2 The Air Traffic Simulator 5
2.1 Air Traffic Management in the Real World 5
2.2 A Model of Airspaces and Air Traffic 6
2.3 DemoSession Data structure 7
2.4 Graphical User Interface 8

3 System Specification 10
3.1 Class organisation 10
3.2 Package `ats.airspace` 11
3.3 Package `ats.timetable` 16
3.4 Package `ats.demo` 21
3.4.1 DemoSession Modelling 21
3.4.2 DemoSession Modelling Described 21
3.4.3 DemoSession Function Description 22
3.4.4 DemoSession Implementation and Package Structure . 23
3.5 Package `ats.gui` 24
3.5.1 XtraToolPanel class with extensions 24
3.5.2 EditorFrame class 25
3.5.3 Other classes in ats.gui package 26
3.6 Package `ats.misc` 26
3.7 Package `ats.animation` 27
3.8 Alternative Projections 28

4 Simulation Scenarios 30
4.1 Case I: Billund Airport 30
4.2 Case II: Random Time Table 33

5 Conclusions 37

A User's Guide 38
A.1 Starting the Program 38
A.2 The WelcomeFrame and Getting Started 39

A.2.1 The File Menu Bar 39
A.2.2 New Session 39
A.2.3 Open Session 40
A.3 The EditorFrame 40
A.3.1 The File Menu Bar 40
A.3.2 The Edit Menu Bar and the Undo/Redo Functionality 41
A.4 The Airspace Module 42
A.4.1 AS Options Menu 42
A.4.2 View Menu 42
A.4.3 Airports 43
A.4.4 Airways 43
A.5 The Time Table Module 44
A.5.1 TT options menu 44
A.5.2 Aircraft 45
A.5.3 Flights 45
A.5.4 Landings 45
A.5.5 Other Options 46
A.5.6 The Table 46
A.6 The Animation Module 46
A.7 The Properties Menu 47
A.7.1 Time Table Properties 47
A.7.2 Airspace Properties 47
A.8 Contact Information 47

B System Verification **49**
B.1 Datastructure Tests 49
B.2 GUI Test 51
B.2.1 Panel test 51
B.2.2 GetAppropiatePanel Method Test 61

C DemoEvent Table **63**

Bibliography **63**

Chapter 1

Introduction

This report documents the development and usage of an air traffic simulation system, in the following abbreviated to ATS.

The simulator visualises and animates flight schedules in real-time or fast-time[1]. The aim of the project has been to create a basic simulation system with a graphical user interface providing the framework for future extensions.

The ATS system provides a tool for flight planners, allowing them to simulate flight schedules and identify problems and possible improvements before applying the schedules to actual traffic.

Traditionally aircraft have navigated by following signals emitted by radio beacons. Now satellite positioning provides a superior means of establishing location, which is independant of ground based systems. This has lead to new ideas of how to conduct air traffic. Instead of having controllers on the ground, pilots could choose their own route. This would require a reliable and fast way of predicting spacing conflicts and, more importantly, resolving them automatically. Much research effort has gone into this area. Several very different approaches have been proposed, e.g. using linear programming [4] or genetic algorithms [3]. One application of the ATS system may be to evaluate and compare the performance of these methods.

Another major issue in air traffic management is the congestion of the airspace over Europe and parts of the US. The ATS system may be used to determine where the network of airways is overloaded and where extra capacity is available.

Chapter 2 describes the system domain, the model used and the user interface. Chapter 3 gives an overview of the components of the software system and includes a formal specification of modules and functions using Standard ML. Two case studies are given in chapter 4. Chapter 5 contains the conclusions and an assessment of the state of the simulator. A user's guide

[1] Fast-time means that e.g. 1 hour in the real world is compressed into 1 minute in the simulator.

is included in appendix A. Tests of the software system are summarized in appendix B. Finally, appendix C contains a table of values identifying events.

Chapter 2

The Air Traffic Simulator

This chapter begins with a brief description of how air traffic is managed today. From this description a model of airspaces and air traffic is derived and the basic principles of the user interface are presented.

2.1 Air Traffic Management in the Real World

The world's airspace is divided into controlled and uncontrolled zones. Aircraft flying within controlled zones are directed by air traffic controllers from ground-based control centres. In contrast, aircraft flying wholly within uncontrolled zones are free to manoeuvre subject only to certain flight rules.

Controlled airspace essentially refers to terminal areas and airways. The airspace in the immediate vicinity of major airports is designated terminal area. No aircraft may enter a terminal area without the permission of the controller responsible for that area. The terminal area serves to separate aircraft either taking off or landing at the airport from other traffic. Typically, terminal areas extend from ground level up to about 24,500 ft. Airways or air routes are corridors through the airspace connecting the terminal areas of different airports.

Most of the world's civil air traffic is currently conducted in controlled zones. However, in the future pilots may have more freedom in choosing their route between airports. In this situation airways are no longer required, while terminal areas are still necessary, because pilots must be able to focus on the landing and not worry about other aircraft getting in the way.

Aircraft flying between two airports must file a flightplan with local authorities some time before takeoff. The flightplan contains details about the flight, the most important being the departure and destination airports and the type of aircraft used. At busy airports the aircraft also needs to be allocated a time for takeoff and landing, a so called slot. During peak hours slots are a scarce resource.

2.2 A Model of Airspaces and Air Traffic

The model of airspaces used in the ATS system follows the general principles of air traffic management outlined above. Terminal areas are called airdomes, while airways retain their name.

An airspace may have a boundary that marks the area of interest. For example if one wants to visualize the traffic passing through the area controlled by a specific control centre, the boundary should be set to the boundary of that area.

An airport is modelled as a point from which aircraft appear and disappear. We distinguish between airports located within the boundary and airports located outside of the boundary. Those within are called internal airports, while those outside are called external airports. Internal airports may have an airdome. External airports do not have airdomes.

Airways consist of a number of sections, each defined by two points on the ground and an interval of altitudes. The line connecting the two points marks the path of the section. The interval determines which altitudes are legal for an aircraft following that section of an airway. The first section must start directly above an airport. Each of the following sections must start where the previous section ended, and the final section must end directly above another airport. Airways allow traffic in both directions.

As with the model of airspaces, the structure used for keeping track of flights follows the principles described in the previous section. We refer to this structure as the *time table*. The time table records for each flight the departure and destination airports, the time of departure and arrival, and the type of aircraft used on the flight. If the flight has intermediate landings, then the corresponding entry in the time table has more than two airports and also stores the time of arrival and departure at the intermediate airports.

Slots are not included in the model, i.e. we assume that an aircraft is allowed to takeoff and land at any given time.

By joining the information in the time table with the airspace, the progression of a flight can be calculated. This is referred to as *scheduling*. The route followed and the altitude of the aircraft are determined automatically and cannot be influenced by the user of the system.

Aircraft will takeoff, follow a series of navigation marks, called *waypoints*, and land. If an airway exists between the departure and destination airports the aircraft will follow it. If no airway exists the aircraft will fly the shortest path between the airports.

The shortest path between two points on a sphere is along part of a great circle. Aircraft will follow great circles and the curves connecting the waypoints of an airway are also great circles.

The model sketched above implies some simplifications in comparison to the real world. Some of these are discussed below.

- No information about the number or orientation of runways is stored.
- Aircraft taking off from an airport will be heading for their first waypoint immediately after takeoff.
- Aircraft coming in for landing will land in the same direction that they followed between the last waypoint and the airport.
- Two or more aircraft are allowed to use an airport simultaneously.
- Aircraft positions are calculated at discrete points in time.
- The ground speed of an aircraft is not affected by the aircraft climbing or descending.
- Fuel comsumption is ignored, although no aircraft will be allowed to fly further than its maximum range.
- Flight delays are not considered.
- Weather conditions are not considered.

The justification for the first four simplifications is that the ATS system is not intended to give a detailed simulation of proceedings at or immediately around an airport. However, in order to detect overloading of an airport or an airway, these are equipped with a maximum capacity. This may be used to issue a warning message whenever the capacity is exceeded, e.g. if the number of takeoffs and landings at an airport within a given time period would be impossible in reality, where separation rules must be observed.

The simulator records the accumulated number of takeoffs and landings in 15 minute intervals for every airport. It also checks for spacing violations, i.e. aircraft that are closer to each other than regulations permit.

The last five simplifications reduce the complexity of the simulator with limited loss of functionality.

2.3 DemoSession Data structure

The `DemoSession` data structure is intended to add a useful functionality to the main program's user interface. It is intended for this structure to allow a user to work with the program and build up information of his work as he goes along. This information is a log of user made changes to the data structure, which we refer to as the *demo sequence*. This demo sequence is then effectively a list of chronologically ordered *events* the user has performed. By event, we mean any change to the time table or airspace data structures.

The user may then move back in the demo sequence if the user regrets certain events, or move forward through them if so desired. If a user moves

back in the demo sequence and creates any new events, the user will automatically have discarded any information about events that may have been ahead of his position in the demo sequence. This helps to prevent inconsistencies in the data the user is working with.

Finally, the user should be able to save his demo sequence and relevant data structures to a file and be able to continue his session later. The user should also be able to save the individual vital data structures separately, which in our case amount to the time table and airspace data structures.

- Only changes in the data structure are saved in the demo sequence. Purely visual changes such as window changes, zoom moves and the like are not monitored.
- The user can move back and forth throughout the entire demo sequence.
- Moving back in the demo sequence and then creating new events will permanently discard any information stored ahead of the users position in the demo sequence when he performed the event.

2.4 Graphical User Interface

In order to make the ATS program easy to survey we present the program through a graphical user interface (GUI). The design for this GUI is modelled on figure 2.1. As in Java, we denote an independent window by a frame. By a panel we mean a container that holds GUI components. As it appears on figure 2.1 we present the program through two frames in which panels are attached whenever it is appropriate. In order to have a figure that illustrates visual changes we have made some conventions. An arrow indicates that the scenery changes. A dotted arrow means that you stay in the same frame, but the panel changes. If the arrow is filled it means that a new frame will appear. The small text in some of the panels gives a brief information about the purpose of the panel. As regards the panels at the two large tags (TIMETABLE and AIRSPACE) they will use methods from each of the time table and airspace data structures to carry out their purpose.

The WelcomeFrame gives a short introduction to the program, and through the frame the user decides how he would like to start his session. This decision could either conclude in starting a new session or opening an old session. After making this decision the EditorFrame is shown. Through the EditorFrame you will be able to manipulate all data structures presented in the ATS program. All panels that deal with editing the data structure of the time table or airspace are presented through this frame. Each method in the time table and airspace data structure, whose intention is to be accessible for the user, has a panel from which the method can be invoked. We

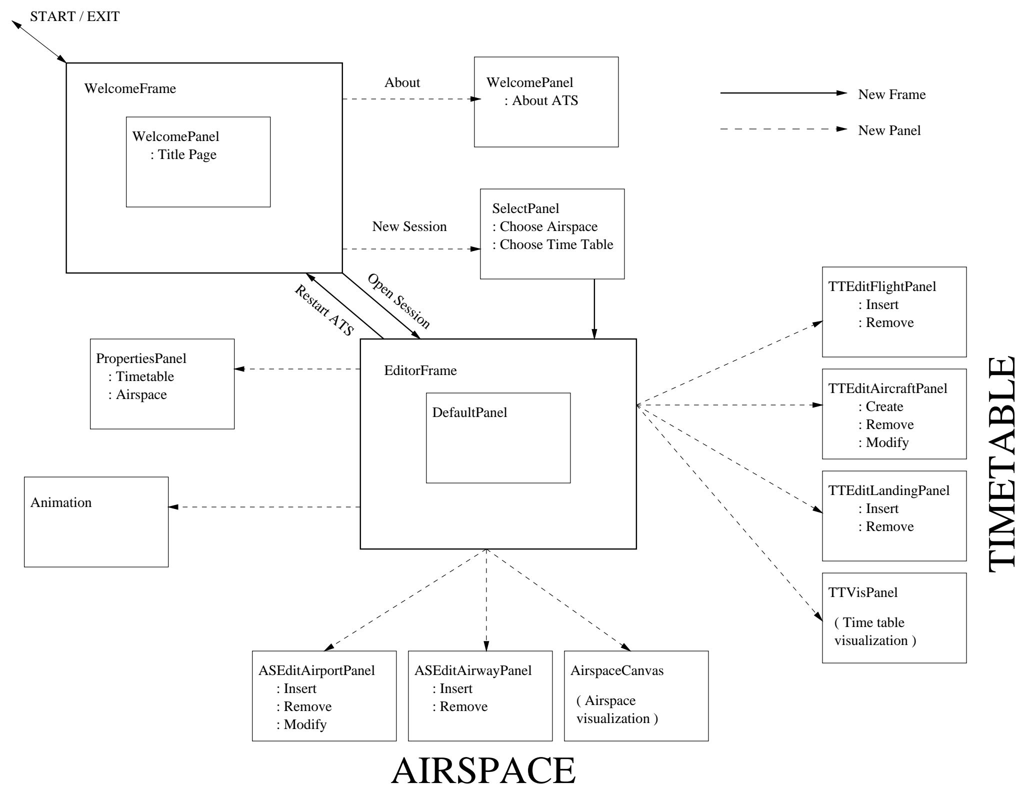

Figure 2.1: Program overview

store all data structure changes, made through the EditorFrame by using the DemoSession functionality.

Chapter 3

System Specification

This chapter aims to give an overview of the software system and the public interface to the underlying data structure.

3.1 Class organisation

The Java classes that comprise the simulator are divided into a number of packages. The package names and short descriptions are listed in table 3.1.

The behaviour of classes that form part of the data structure is specified using Standard ML (SML) signatures. The behaviour of the graphical user interface is specified using figures and short descriptions.

For more detailed documentation of the classes and methods of the ATS system, refer to the Javadoc documentation on the ATS homepage. See appendix A for details of how to access the documentation.

In order to increase clarity and more closely follow the structure of the Java code, the SML syntax and semantics are somewhat abused. SML types will correspond to Java classes, SML datatypes are used where class inheritance is important in the Java code, and overloading of SML functions is allowed. Additionally Java core classes and primitive types are assumed to be valid SML types, e.g. String, int and float.

Many of the classes documented below have the methods `hashCode`, `clone` and `equals`. Because these work the same way for all classes, they are documented here.

The method `hashCode` returns an integer value that is guaranteed to be the same for all objects that are equal according to the method `equals`, which is defined in all classes that implement `hashCode`. The method is used exclusively as a hash function when inserting objects into hash tables.

The `clone` method creates a copy of the object it is invoked on. The copy is initialized to the same state as the original object, however, no references are shared by the two objects, ensuring that changes to one object does not affect the other object.

Package	*Description*
`ats.airspace`	Classes used for the internal representation of airspaces, including airports, airways etc.
`ats.timetable`	Classes for time tables.
`ats.demo`	Classes for representing demo sequences.
`ats.gui`	Contains the graphical user interface.
`ats.misc`	Miscellaneous classes for storing system configuration and for testing.
`ats.animation`	Contains classes used for the animation.

Table 3.1: Air Traffic Simulator Java packages.

Equality of objects is defined by `equals`. The criteria for equality depends on the class, however, for most classes `equals` returns true if and only if the internal fields have the same values in the two objects being compared. In the documentation below, `equals` methods are only commented for classes that do not conform to this definition.

3.2 Package `ats.airspace`

The classes of this package may be divided into two groups; one containing the classes used for storing airspaces with their associated airports, airways etc., and one containing the classes used to draw airspaces on the screen. The classes of the first group are documented first.

Airways, airdomes and airports are equipped with a capacity indicating how much traffic can be handled in a given time interval. This capacity is represented by a `Capacity` object. The class has a method (`exceedsCapacity`) for comparing two capacities to determine if one exceeds the other.

```
type Capacity = int
val Capacity        : int → Capacity
val changeCapacity  : Capacity × int → Capacity     (* modifies capacity *)
val clone           : Capacity → Capacity
val exceedsCapacity : Capacity × Capacity → boolean
val getCapacity     : Capacity → int
```

An object of the class `Name` represents the name and identification code (ID) of an airport. The name is an arbitrary length string, while the ID is an upper case string, usually composed of three letters. The ID is required to be unique within an airspace.

The class has two constructors, one takes a single string containing both name and ID in a special format compatible with `toString`. The other constructor takes the name and ID as separate strings.

```
type Name = String × String                                  (* name × id *)
```

```
val Name            : String × String → Name
                  (*  name      id  *)
val Name            : String → Name
val clone           : Name → Name
val equals          : Name × Name → boolean
val getID           : Name → String
val getName         : Name → String
val hashCode        : Name → int
val toString        : Name → String
```

The width of an airway is stored in an object of the class `Width`.

```
type Width = float
val Width           : float → Width
val changeWidth     : Width × float → Width        (* modifies the width *)
val clone           : Width → Width
val getWidth        : Width → float
```

Locations are represented by objects of type `Point`. `Point` itself is an abstract class and provides methods common to points in spaces of arbitrary dimension. The classes `Point1`, `Point2` and `Point3` are subclasses of `Point` and correspond to points in spaces of dimension one, two and three, respectively. `Point1` is just a scalar value intended for handling coordinates of a `Point2` or `Point3` separately.

Assume a point P_3 has coordinates (p_1, p_2, p_3). Calling `convertDimension` with arguments P_3 and 2 yields a new point P_2 in 2-space with coordinates (p_1, p_2). Calling `convertDimension` with P_2 and 3 as arguments yields a point in 3-space with coordinates $(p_1, p_2, 0)$.

The method `equals` defines equality of points. Two points are equal if and only if they are contained in spaces of the same dimension and have the same coordinates.

```
datatype Point = Point1 of float
               | Point2 of float × float
               | Point3 of float × float × float
val Point1          : float → Point1
val Point2          : float × float → Point2
val Point3          : float × float × float → Point3
val clone           : Point → Point
val convertDimension: Point × int → Point
val equals          : Point × Point → boolean
val getCoordinate   : Point × int → float        (* a single coordinate *)
val setCoordinate   : Point × int × float → Point     (* changes coord. *)
```

An interval of altitudes is represented by an object of the class `Interval`. The method `contains` checks whether a given value is contained in the interval.

```
type Interval = float × float
val Interval        : float × float → Interval
```

```
val clone          : Interval → Interval
val contains       : Interval × float → boolean
val getMax         : Interval → float          (* returns upper bound *)
val getMin         : Interval → float          (* returns lower bound *)
```

An area on the surface of the Earth may be represented by a `Polygon` object. The method `contains` checks whether a given point is contained in the interior of the polygon. In this implementation the polygon is not well-defined because on a sphere any closed curve will surround two areas. More specifically, `contains` cannot be relied on if both areas include a pole.

```
type Polygon = Point2 list
val Polygon        : unit → Polygon
val Polygon        : Point2 list → Polygon
val addPoint       : Polygon × Point2 → Polygon        (* adds vertex *)
val clone          : Polygon → Polygon
val contains       : Polygon × Point2 → boolean
val getPoints      : Polygon → Point2 list        (* returns vertices *)
```

The special airspace surrounding internal airports is stored as an `Airdome` object. An `Airdome` is specified by a polygon marking the base, an interval of altitudes marking the floor and ceiling and a capacity indicating the maximum allowable flow of traffic. `exceedsCapacity` will check whether this capacity has been exceeded. The `contains` method determines whether a point is inside the air dome.

```
type Airdome = Polygon × Interval × Capacity
val Airdome         : Polygon × Interval × Capacity → Airdome
val clone           : Airdome → Airdome
val contains        : Airdome × Point3 → boolean
val exceedsCapacity : Airdome × Capacity → boolean
val getCapacity     : Airdome → Capacity
val getInterval     : Airdome → Interval
val getPolygon      : Airdome → Polygon                (* base polygon *)
```

Airports are specified as `Airport` objects containing a location, a name and an ID. Airports are equal if their `Name` objects are equal.

```
type Airport = Point3 × Name
val Airport        : Point3 × Name → Airport
val clone          : Airport → Airport
val equals         : Airport × Airport → boolean
val getLocation    : Airport → Point3
val getName        : Airport → Name            (* airport name and ID *)
val hashCode       : Airport → int
val toString       : Airport → String        (* string representation *)
```

An internal airport differs from an airport by having an airdome and a capacity associated with it. The airdome should be positioned directly above the internal airport, but this restriction is not enforced. The capacity

is a measure of the maximum number of takeoffs and landings possible in a given time interval. `exceedsCapacity` can check if this capacity has been exceeded.

```
type InternalAirport = Point3 × Name × Airdome × Capacity
val Internalairport : Point3 × Name × Airdome × Capacity → InternalAirport
val clone           : InternalAirport → InternalAirport
val equals          : InternalAirport × InternalAirport → boolean
val exceedsCapacity : InternalAirport × Capacity → boolean
val getCapacity     : InternalAirport → Capacity
val getDome         : InternalAirport → Airdome
val getLocation     : InternalAirport → Point3
val getName         : InternalAirport → Name              (* name and ID *)
val hashCode        : InternalAirport → int
val toString        : InternalAirport → String
```

The path of an airway is specified by a list of n points where $n \geq 2$. Two successive points mark a section of the airway and each section has a base and a ceiling altitude. Additionally an airway has a width and a capacity common to all sections. The capacity is a measure of the maximum allowable flow of traffic through the airway. The method `exceedsCapacity` checks if this capacity has been exceeded. The `contains` method determines whether a point is in the volume of the airspace where the airway exists.

```
type Airway = Width × Capacity × Point2 list × Interval list
val Airway          : Width × Capacity × Point2 list × Interval list
                                                                → Airway
val clone           : Airway → Airway
val contains        : Airway × Point3 → boolean
val exceedsCapacity : Airway × Capacity → boolean
val getCapacity     : Airway → Capacity
val getIntervals    : Airway → Interval list       (* altitude intervals *)
val getLength       : Airway → float           (* total length of airway *)
val getPoints       : Airway → Point2 list             (* base waypoints *)
val getWidth        : Airway → Width
```

All airports and airways of an airspace are contained in an object of the class `Airspace`. A special part of the airspace may be designated by a base polygon and an interval of altitudes.

```
type Airspace = Polygon × Interval × Airport list × Airway list
val Airspace        : Polygon × Interval → Airspace
val clone           : Airspace → Airspace
val contains        : Airspace × Point3 → boolean
val createAirport   : Airspace × Airport → Airspace      (* adds airport *)
val createAirway    : Airspace × Airway → Airspace        (* adds airway *)
val distance        : Point2 × Point2 → float        (* distance between *)
val distance        : Point3 × Point3 → float        (*  two locations   *)
val getAirport      : Airspace × Name → Airport
    (* returns airport with given name *)
val getAirways      : Airspace × Name × Name → Airway list
```

```
    (* returns airways between two airports *)
val getBounds        : Airspace → Polygon
    (* returns airspace bounds *)
val getExternalAirports : Airspace → Airport list
    (* returns all external airports *)
val getInternalAirports : Airspace → InternalAirport list
    (* returns all internal airports *)
val getAirports      : Airspace → Airport list
    (* returns all airports *)
val isDefinedAirport: Airspace × Name → boolean
    (* true if airport exists *)
val isExternalAirport : Airspace × Name → boolean
    (* true if airport exists and is external *)
val isInternalAirport : Airspace × Name → boolean
    (* true if airport exists and is internal *)
val moveAirport      : Airspace × Name × Point3 → Airspace
    (* moves airport to new location *)
val removeAirport    : Airspace × Name → Airspace
    (* removes airport from airspace *)
val removeAirway     : Airspace × Airway × Name × Name → Airspace
    (* removes airway from airspace *)
```

Next follows the classes used for drawing an airspace on the screen.

The mapping from longitude/latitude to screen coordinates is handled by a subclass of the abstract class `Projection`. Every subclass contains information about the type of projection it provides, the primary features of that projection and the system properties required by the class. This information is available through `getType`, `getDescription` and `getProperties` respectively.

The ATS system is supplied with two projections, namely `CylindricalProjection` and `ConformalConicalProjection` both of which reside in the `ats.airspace` package.

Note that `AWTPoint` is short for `java.awt.Point`.

```
type Projection
val Projection       : unit → Projection
val clear            : Projection → unit                  (* clears image *)
val drawCircle       : Projection × Point2 × Point1 → unit
val drawGeodesic     : Projection × Point2 × Point2 → unit
    (* draws part of a great circle *)
val drawGeodesic     : Projection × Point3 × Point3 → unit
    (* draws part of a great circle *)
val drawGrid         : Projection → unit
val drawLine         : Projection × Point2 × Point2 → unit
val drawLine         : Projection × Point3 × Point3 → unit
val drawPoint        : Projection × Point2 → unit
val drawPoint        : Projection × Point3 → unit
val drawPolygon      : Projection × Polygon → unit
val drawString       : Projection × String × Point2 → unit
val drawString       : Projection × String × Point2 × int → unit
```

```
val getCoordsFor    : Projection × AWTPoint → Point2  (* inverse mapping *)
val getDescription  : Projection → String
val getGraphics     : Projection → Graphics
val getProperties   : Projection → String
val getType         : Projection → String
val keepAspectRatio : Projection × boolean → Projection
val projectPoint    : Projection × Point3 → AWTPoint
    (* projects point without drawing *)
val setBackground   : Projection × Color → unit
    (* changes background color *)
val setBounds       : Projection × Point3 × Point3 → Projection
val setCenter       : Projection × Point2 → Projection
val setColor        : Projection × Color → unit    (* sets drawing color *)
val setGraphics     : Projection × Graphics × Dimension → Projection
val zoom            : Projection × Point2 × float → Projection
    (* zooms in/out (depending on the value of the float argument) *)
```

The class `AirspaceRenderer` determines how airports and airways look on the screen.

```
type AirspaceRenderer
val AirspaceRenderer: Airspace → AirspaceRenderer
val highlight       : AirspaceRenderer × Object → AirspaceRenderer
    (* highlights an object so it is drawn in another color *)
val render          : AirspaceRenderer × Projection → unit
    (* draws airspace *)
```

`AirspaceCanvas` uses all the previously described classes to present an airspace on the screen. The image will automatically update in response to changes in the underlying airspace.

```
type AirspaceCanvas
val AirspaceCanvas  : Airspace → AirspaceCanvas
val AirspaceCanvas  : Airspace × Projection → AirspaceCanvas
val getCoordsFor    : AirspaceCanvas × AWTPoint → Point2
val highlight       : AirspaceCanvas × Object → unit
val setBounds       : AirspaceCanvas × Point3 × Point3 → unit
val setCenter       : AirspaceCanvas × Point2 → unit
val zoom            : AirspaceCanvas × Point2 × Float → unit
```

3.3 Package `ats.timetable`

A time table is used to schedule flights. It stores information about which airports the various flights fly to, and at which arrival and departure times. The time table also stores all the aircraft types that are available for the flights. The available airports for the time table are taken from the airspace supplied, when constructing the time table.

The main class of the package is `TimeTable`. It contains query and modifying methods (e.g. `containsFlight` and `insertFlight`). The package also

contains some classes that are used to view and edit the time table. It is only the two public classes (`TimeTableApplet` and `AirportTimeTableApplet`) that will be modelled. Finally the package contains some basic types (`Aircraft`, `Flight`, `Time` and `FlighthData`) that are used when modifying the time table. They will be described first.

The `Aircraft` class represents an aircraft type and the information associated with it. Since the `Aircraft` class only models aircraft types, and not concrete aircraft, an `Aircraft` object can be used on two different flights at the same time. If the `Aircraft` object is used in an air traffic simulation system, the units specified below the constructor must be used.

```
type Aircraft
val Aircraft        : String × float × float × float × float ×
                      float × float × float × float → Aircraft
    (* aircraftType  maxSpeed (knots)  cruiseSpeed (knots)  takeoffLandingSpeed (knots)
       maxVertSpeed (feet/min)  range (km)  maxTurnRate (degrees/min)
       maxAcceleration (knots/min)  preferredAltitude (feet)  *)
val equals          : Aircraft → boolean
val getType         : Aircraft → String
val getMaxSpeed     : Aircraft → float
val getCruiseSpeed  : Aircraft → float
val getTakeoffLandingSpeed : Aircraft → float
val getMaxVertSpeed : Aircraft → float
val getRange        : Aircraft → float
val getMaxTurnRate  : Aircraft → float
val getMaxAcceleration : Aircraft → float
val getPreferredAltitude : Aircraft → float
val toString        : Aircraft → String
```

A `Flight` consists of a unique flight number and an aircraft to be used on the flight.

```
type Flight
val Flight          : String × Aircraft  → Flight
    (* The flight number is represented by a string *)
val clone           : Flight → Flight
val equals          : Flight → boolean
val getAircraft     : Flight → Aircraft
    (* Returns the aircraft used on the Flight. *)
val getFlightNumber : Flight → String
    (* Returns the flight number of the Flight. *)
val hashCode        : Flight → int
val toString        : Flight → String
```

The type `Time` represents a specific instant in time with precision down to seconds.

```
type Time
val Time            : int × int × int × int × int → Time
                  (*  year   month  date   hour   minute *)
val Time            : int × int × int × int × int × int → Time
```

```
                  (*  year   month  date   hour   minute second *)
val add             : Time × float  → Time
    (* Adds the specified (signed) amount of seconds to the Time. *)
val after           : Time × Time → boolean
    (* Tests if the first Time is after the second Time. *)
val before          : Time × Time → boolean
    (* Tests if the first Time is before the second Time. *)
val clone           :  Time → Time
val compareTo       :  Time × Time → int
    (* Compares the first Time to the second Time. The method returns
       the value 0 if the first Time is equal to the second Time;
       a value less than 0 if the first Time is before the second Time;
       and a value greater than 0 if the first Time is after the second Time.*)
val equals          : Time × Time → boolean
val fromString      : String → Time
    (* Converts a string into a Time. *)
val getCurrentTime  : Time
    (* Creates a Time using the current time. *)
val timeDifference  :  Time × Time → double
    (* Returns the difference between two times, measured in minutes. *)
val toString        :  Time → String
```

When constructing an illegal `Time` (e.g. Mon Mar 32 28:22 2000), an exception is thrown:

```
exception IllegalTimeException
exception IllegalTimeException of String
```

A `FlightData` object represents a landing in a time table. It stores information about the airport, arrival time and departure time. If the FlightData object represents the first departure on the route of the flight, the arrival time should be `null`. Similarly, the departure time should be `null`, if the landing is the last landing on the route of the flight.

```
type FlightData
val FlightData      : Airport × Time × Time → FlightData
    (* Constructs a value of type FlightData with the specified airport,
       arrival and departure time, respectively. *)
val clone           : FlightData → FlightData
val equals          : FlightData × FlightData → boolean
val getAirport      : FlightData → Airport
val getArrival      : FlightData → Time
val getDeparture    : FlightData → Time
val toString        : FlightData → String
```

The operations on a time table can be grouped into two categories: queries, which return information about the time table, and modifying operations. In addition, the `TimeTable` class contains a few public auxiliary methods. The modifying methods will be documented first.

Many of the modifying operations return a boolean. They return true if the modification succeeded, and false if the time table was not modified due to some error.

The modifying methods `replaceFlight`, `replaceAircraft`, `insertLanding`, `replaceAirport`, `setTime` and `setMinTimeSeperation` might violate the assumption that an aircraft has to be in the airport for a minimum of time before taking off again. Postponing the departure and arrival times in question circumvents this.

```
type TimeTable  = (Flight, FlightData vector) Table.table
val TimeTable       : Airspace → TimeTable
val insertFlight    : TimeTable × Flight × FlightData vector → TimeTable ×boolean
    (* Maps the specified Flight to the specified FlightData vector in the time table. *)
val replaceFlight   : TimeTable × Flight × Flight → TimeTable × boolean
    (* Replaces a flight (second argument) with an other one (third argument) *).
val removeFlight    : TimeTable × Flight → TimeTable × boolean
val createAircraft  : TimeTable × Aircraft → TimeTable × boolean
val replaceAircraft : TimeTable × Aircraft × Aircraft → TimeTable
val removeAircraft  : TimeTable × Aircraft → TimeTable × Aircraft
val insertLanding   : TimeTable × Flight × Airport × Time → TimeTable
    (* Inserts a landing for a flight with the specified airport and departure time. *)
val removeLanding   : TimeTable × Flight × Airport × Time → TimeTable ×boolean
    (* Removes the landing in the airport at the specified departure
       time from the flight route. *)
val replaceAirport  : TimeTable × Flight × FlightData × Airport → TimeTable× boolean
    (* Replaces the airport in the FlightData object with the given airport. *)
val setTime         : TimeTable × Flight × FlightData × Time → TimeTable
    (* Sets the departure time in the flight's specified FlightData object. *)
val setMinTimeSeparation : TimeTable × TimeTable × float
    (* Sets the minimum time separation between an arrival
       and a departure on a flight. *)
```

The query and public auxiliary methods in the `TimeTable` class are the following:

```
val getAircraft     : TimeTable × String → Aircraft
    (* If the argument aircraft type (the string) is registered in the time table,
       the corresponding Aircraft object is returned. *)
val getAirports     : TimeTable → Airport vector
    (* Returns a vector containing all the airports used by the time table. *)
val getAllAircraftTypes : TimeTable → Aircraft vector
    (* Returns all the aircraft types used by the time table. *)
val getArrivals     : TimeTable × Flight × Airport → Time list
    (* Returns the arrival times to an airport on a flight. *)
val getDepartures   : TimeTable × Flight × Airport → Time list
    (* Returns the departure times to an airport on a flight. *)
val getFlight       : TimeTable × String → Flight
    (* If there exists a flight with the argument flight number, this flight is returned. *)
val getFlightData   : TimeTable × Flight → FlightData list
    (* Returns an array containing the FlightData to which the
       specified flight is mapped in the time table. *)
val getMinTimeSeparation : TimeTable → float
    (* Returns the required minimum time separation between an arrival
       and a departure of a flight. *)
val getTimeTable    : TimeTable → Object list list
    (* Inserts the contents of the time table in a two dimensional array. *)
```

```
val isEmpty          : TimeTable → boolean
val containsFlight   : TimeTable × Flight → boolean
val numberOfFlights : TimeTable → int
val calculateArrival: TimeTable × Aircraft × Time × Airport × Airport → Time
   (* Calculates the arrival time of an aircraft, when it departs at the specified
      departure time and flies between the two airports. *)
val clone            : TimeTable → TimeTable
val toString         : TimeTable → String
val write            : TimeTable × String → unit
    (* Writes the content of the time table to a file named fileName. *)
```

Some operations in the time table requires special exceptions:

```
exception AircraftRemovalException
exception AircraftRemovalException of String
    (* Thrown when an attempt has been made to remove an aircraft in use. *)
exception IllegalRangeException
exception IllegalRangeException of String
    (* Thrown to indicate that the range of an aircraft is to small,
       compared with the distance it is supposed to fly. *)
```

The contents of a time table can be seen by means of the `toString` or `write` methods. It can also be seen by inserting it into an applet.

```
type TimeTableApplet
val TimeTableApplet : TimeTable → TimeTableApplet
```

When wrapped in an applet, the time table becomes partly editable:

The aircraft type and flight number of a flight can be replaced, when clicking on the flight. This is done in a dialog window, where the data of the aircraft also are displayed. An airport on the route of a flight can be replaced by clicking on the airport. The options are displayed in a drop-down list. The departure time is edited by typing the new departure time in the text field. If the new time is invalid or causes some conflict in the time table, an error message is displayed. Other modifications of the time table are performed by means of menu items in the menu bar. The classes `FlightCellEditor`, `AirportCellEditor`, and `TimeCellEditor` maintain the three sorts of modifications mentioned above. They all extend the class `javax.swing.DefaultCellEditor` and overrides the methods `fireEditingStopped`, `getCellEditorValue`, and `getTableCellEditorComponent`. They will not be modelled in SML, since they are only auxiliary classes for the `TimeTableApplet` class and therefore only have package access.

The `TimeTable` (and `TimeTableApplet`) emphasises the flights. If the user wants to see a table that emphasises the traffic in and out of the various airports, this can be done by means of an `AirportTimeTableApplet`. In an `AirportTimeTableApplet` the first column contains the airports in the time table and the second column contains the flights. Like in the ordinary time table, the third and fourth column contains the arrival and departure times, respectively. This reordering makes it possible to see when the flights enters

and leaves an airport, and at which times. Unlike the TimeTableApplet, the AirportTimeTableApplet is not editable.

```
type AirportTimeTableApplet
val AirportTimeTableApplet : TimeTable → AirportTimeTableApplet
```

3.4 Package `ats.demo`

As mentioned earlier, the DemoSession data structure exists to keep track of the *events* the user has performed upon the time table or airspace data structures. These events are collected in order in what we call a *demo sequence*. This section deals with this functionality of the ATS program.

3.4.1 DemoSession Modelling

```
val TIMETABLE : TimeTable (* These two values are CONSTANTS from the start *)
val AIRSPACE  : Airspace  (* and never altered throughout the session *)

val TT : TimeTable     (* These two values are updated normally      *)
val AS : Airspace      (* to reflect the events user has performed   *)

type Method            (* information on a specific method           *)
type EventID = int     (* each event has its own ID                  *)
type Index = int       (* where we are in the DemoItem list          *)
val NewDataStructure   (* encapsulates updated versions of AS and TT *)

datatype V = int | String | boolean |  ... (* actual values *)

type DemoEvent = Method
type DemoItem = DemoEvent × EventID × V list
type Demo_sequence = DemoItem list
type DemoSession = TimeTable × Airspace × Demo_sequence

val recreate : DemoItem → NewDataStructure
val previous : DemoSession × Index → NewDataStructure
val next     : DemoSession × Index → NewDataStructure
val start    : DemoSession → NewDataStructure
val end      : DemoSession → NewDataStructure

val save     : DemoSession × int → File (* int : what to save *)
```

3.4.2 DemoSession Modelling Described

The essential arguments for the `DemoSession` data structure is a TimeTable and an Airspace. When provided, these two are cloned where the originals are named `TIMETABLE`, and `AIRSPACE` for the TimeTable and Airspace respectively. These two are never altered and serve as a starting point for operations concerning the demo sequence. The clones are named `TT` and `AS`. These two are then modified as the user goes along his business in his session.

To complete the data structure we need to model how the demo sequence is made.

Any action by the user that alters the data structure at hand is known as a `DemoEvent`. To keep a track of the `DemoEvent`s we specify the following data types: `Method` for the method or function used to carry out the Event (here we named it Method since we implemented our program in the Java language), `EventID` as a unique ID number to identify the Event that took place (see appendix C for a list of ID numbers and their description), and `V list` for a list of values used by the method that carried out the changes in the relevant event.

Furthermore, the information surrounding an event is collected into a data structure we call `DemoItem`. This structure combines `Method`, `EventID` and `V list`. The demo sequence is in effect a `DemoItem list`.

To keep a track of where the user is actually in his demo sequence when he is in his session, we use `Index` as a pointer to show us where we are.

3.4.3 DemoSession Function Description

The DemoSession program comprises several functions which the user has direct access to and when performed, modify the data structure values in the ATS program and generates visual output. What is important to note is that the following functions do not alter the data structure that handles the information concerning the demo sequence.

The `recreate` function takes as its argument a single event in the demo sequence. It then effectively recreates the changes in the data structure that took place in this event, as if the user had executed them.

The `next` function performs an event a user would expect from pressing a redo button when working with the program. This takes the user one step forward in the demo sequence, and the program performs the appropriate event. This function can be performed until the user is at the end of the demo sequence.

The `previous` function performs much like the `next` function except that it moves in the opposite direction. It performs an action a user would associate with an undo action. This function can be performed until the user is at the beginning of the demo sequence.

The `end` function takes the user all the way to the end of the demo sequence, running through the events stored regardless of his position in the demo sequence.

The `start` function acts much like the `end` function, except that it takes the user to the beginning of the demo sequence.

The `save` function is designed to allow the user to do one of three things: save an entire DemoSession, save the airspace data or save the time table data.

3.4.4 DemoSession Implementation and Package Structure

It is relatively easy to see how the demo sequence is built up as the user moves along in his session. However, it is an interesting question what exactly happens when the user moves back in the sequence. To solve this problem, we chose a rather simple solution.

When the user goes back a step, we implemented it such that the program resets the `TT` and `AS` values back to their constants set at the start of the user's session: `TIMETABLE` and `AIRSPACE`, and then moves forward through the demo sequence step by step, using `recreate` to change the data structure as the user has done until it reaches the `DemoItem` previous to his original position in the sequence.

Moving forward in the demo sequence is relatively easier, as we simply call `recreate` on the next `DemoItem` in the sequence.

Another interesting question is what happens if the user tries to move too far forward or backward in the sequence? This is handled in Java by creating two separate Exceptions which are thrown. These are then detected by the ATS program and visual cues are shown informing the user of an error.

We implemented `DemoSession` in the Java language as a separate package in our ATS program. This package is identified as : ats.demo and it is here we find all our related files of which there are six.

`DemoEvent` : This class is used to be able to save all the information regarding any method called that alters the data structure. When we have to update our `DemoSession` with a new event, we save information about what method was called that altered the data structure by creating a `DemoEvent` object.

`DemoItem` : This class models individual entries in the demo sequence, which include the following data structures: Information about what method was called to effectuate change in `DemoEvent`, information identifying what exact event took place in `EventID` and a list of values which served as arguments for change in `V list`.

`DemoSession` : This is where most of the functionality of our package is implemented. Here we find the constructor that helps us create `DemoSession` objects, as well as all the functions specified in the modelling previously as well as other functions used internally.

`DemoSave` : In order to make it easy to save a `DemoSession` we collect the 3 objects associated with it together into a single object. The three objects collected is the `DemoSession` data structure itself, the Airspace `AS` object and the TimeTable `TT`.

`EndException` and `StartException` : In creating the demo sequence as a `DemoItem list` and giving him control over moving back and forth in the list, it is clear that we must have some control over what happens when he attempts to move too far back or forward in the sequence. To cover this, we create these Exceptions which are thrown internally in the program, and

which manifest themselves as error dialog boxes in the ATS program itself, informing the user that he has reached the limit of movement in his demo sequence.

3.5 Package `ats.gui`

The grahical user interface is constructed by using the Java Abstract Windowing Toolkit (AWT). The GUI modelling illustrated by figure 2.1 is directly translated into Java classes by extending classes from AWT. The names of theses classes are somewhat similar to the conventions made in the modeling phase. Switching between panels and frames, indicated by arrows on figure 2.1, is implemented by using the java.awt.Menubar system. Each arrow on figure 2.1 correspond to a link in the menubars attached to the two frames.

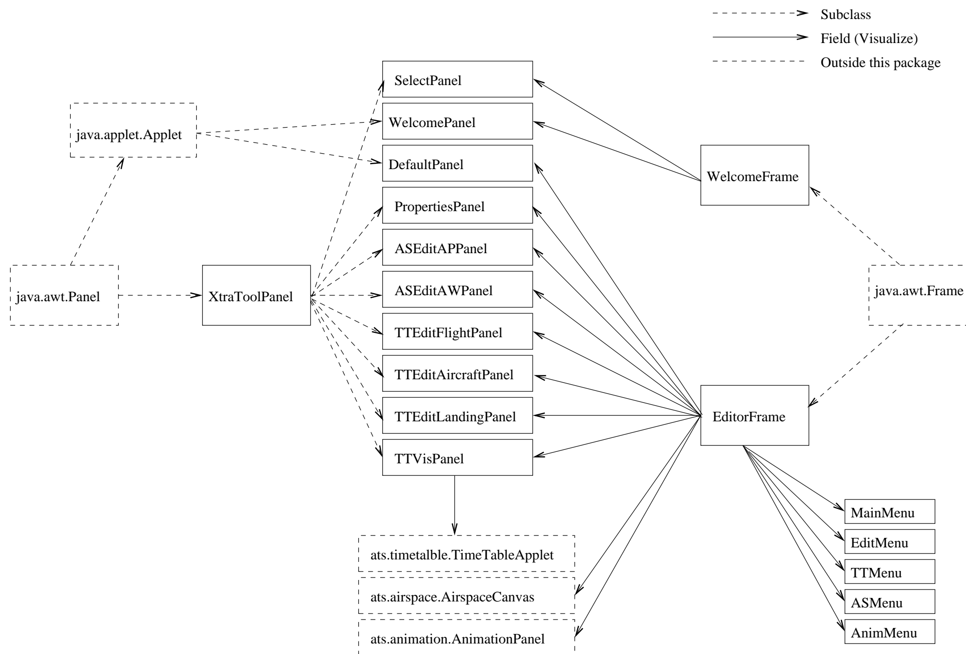

Figure 3.1: Relationship between classes in ats.gui package

3.5.1 XtraToolPanel class with extensions

Most of the panels in this program have the `XtraToolPanel` class as super class. The XtraToolPanel class contains methods that are very useful when contructing a panel. By having the `XtraToolPanel` class as super class we achieve ease in panel creation because of method inheritance. The purpose of

the XtraToolPanel class is having a class that collects all common methods from all the panel classes.

In the XtraToolPanel class you'll find the method reset. This method is declared abstract which makes it possible to construct a reset method for each panel, that extends the XtraToolPanel class. This is convenient because the purpose of the reset method is the same for all panels, but its implementation may vary between each panel. Moreover there are methods to handle simple communication between the user and the panel in the shape of dialogs. These dialogs are imported from the javax.swing package.

The panels that extend the XtraToolPanel class has a common structure.

- AWT components and event listeners are added through the constructor.
- Event handling (by which we mean the handling of events in the traditional AWT sense) is provided through an inner class that distinguish these events by an identification number passed through the inner class's constructor. The event handling works typically in four stages. Firstly, all values are extracted from the panel. Secondly, the values' syntax are checked. Thirdly, the values are stored by using the ats.demo package. Finally, a method is invoked with these values.
- The reset method is defined so it suits the specific panel.

3.5.2 EditorFrame class

In this class you'll find several inner classes that treat events from the menubar. Notable methods in this class are the changeMenu and getAppropiatePanel methods.

The changeMenu method sets the menubar, on this frame, by using at least one of the classes MainMenu, EditMenu, TTMenu, ASMenu or AnimMenu. Each of these classes contains premade menus for a part of the program e.g. the ASMenu is set whenever the airspace data structure is intended to be manipulated.

The purpose of the getAppropiatePanel method is to show a panel that matches the current position in the demo sequence. The method provides appropiate panels for all the events found in appendix C. For each event there is two kinds of an appropiate panel, depending on whether the panel should be shown as it were just before or after performing the event. When it is before the event the panel is shown with the old values as if the user had returned to the screen where he has performed the event. This is done by extracting the values from the current element in the demo sequence into the panel. When it is after the event a new panel is simply initialized. The method is used each time the user changes his position in the demo sequence.

3.5.3 Other classes in ats.gui package

In the `ats.gui` package you'll find some classes which aren't directly related to the hierarchy on figure 3.1. The purpose of these classes are to have some useful functionalities when creating GUI enviroments.

The `AtsFileFilter` class makes it possible to create a filter when choosing files with e.g. the `javax.swing.JFileChooser`. The `JFileChooser` is a premade dialog that is very handy when the user should select a file with belonging path. When the filter is added on the `JFileChooser` it shows only files with a specific extension that is defined through the constructor.

Through the `MouseClickListener` class we can create an object that is able to register all mouse events on an airspace visualization (either `ats.airspace.AirspaceCanvas` or `ats.animation.AnimationCanvas`). The class is used for the zoom methods from the `ats.airspace` package. To enable the zoom methods correctly we need to know where the mouse event is performed on the airspace visualization. This information is provided by the `MouseClickListener` object.

The `ResetHandler` class is able to reset any subclass of `XtraToolPanel` by executing the `reset` method defined in the subclass. This is possible because the `reset` method in the `XtraToolPanel` is declared abstract.

The `WelcomePanel` and the `DefaultPanel` are panels that we use for showing ordinary pictures. The reason why they are extending the `java.applet.Applet` class is that an applet has excellent facilities for visualizing graphics such as pictures.

3.6 Package `ats.misc`

The `ats.misc` package contains classes used in testing the system and a class used to store custom settings affecting the appearance and behaviour of the graphical user interface.

For information about the test classes, refer to appendix B.

The custom settings are maintained by the class `ATSConfig`. All methods of this class are declared `static`, meaning that they operate on the class rather than an instance of the class. Consequently, the class has no constructor.

A setting may be modified by `setProperty`. The value of a setting may be retrieved by using one of `getBoolean`, `getColor`, `getFloat`, `getInt` or `getString`. The choice of method depends on how the value should be interpreted. For example the boolean value "true" must be converted to a string representation in order to store it in `ATSConfig`. To retrieve the value one could use `getString` and get the string representation. However, it would be far easier to use `getBoolean`, which converts the string to the boolean value "true". Attempting to retrieve the value as a color, float or int will result in an exception.

```
type ATSConfig
val ATSConfig        : ATSConfig
val setProperty      : ATSConfig × String → ATSConfig
val getBoolean       : ATSConfig → boolean
val getColor         : ATSConfig → Color
val getFloat         : ATSConfig → float
val getInt           : ATSConfig → int
val getString        : ATSConfig → String
val save             : ATSConfig → unit       (* Saves settings to a file *)
```

3.7 Package `ats.animation`

The animation is run as two concurrent threads. One thread calculates the positions of all active aircraft. This thread is referred to as the *scheduling thread*. The other thread periodically updates the image displayed on the screen to reflect the new calculated positions. This thread is called the *animation thread*.

The threads synchronize and communicate through a buffer implemented as a monitor[1]. The buffer has a fixed capacity. The scheduling thread delivers snapshots of the active traffic to the buffer, and these snapshots are then fetched by the animation thread. If the scheduling thread arrives when the buffer is full, it is forced to wait until the animation thread has fetched a snapshot, thus freeing a slot in the buffer. Conversely, if the animation thread attempts to read from an empty buffer, it must wait until the scheduling thread delivers a snapshot.

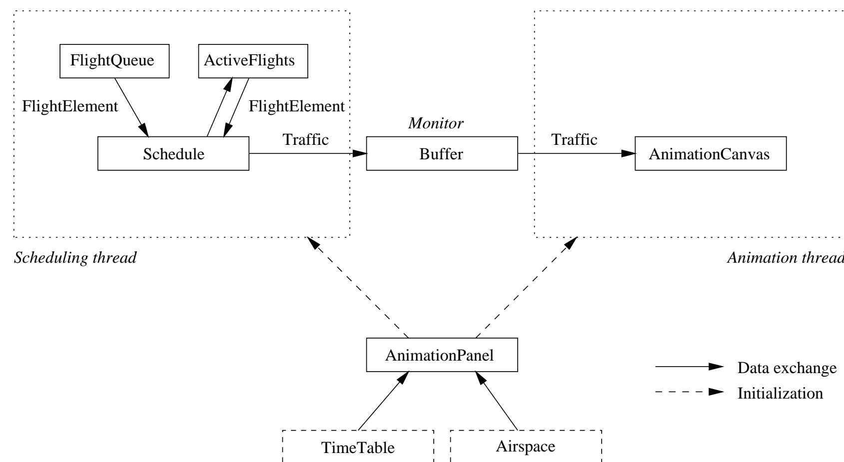

Figure 3.2: Internal structure of the animation package.

The internal organization of the animation module is shown in figure 3.2.

[1] A monitor is a mechanism for ensuring mutual exclusion between concurrent processes, such that only one process at a time is granted access to the variables of the monitor.

The animation is started by initializing an instance of `AnimationPanel` with a time table and an airspace. The flights in the time table are inserted into a priority queue (`FlightQueue`), from which they may be extracted in ascending order of takeoff time. The panel sets up the two threads and waits for the user to start the animation by pressing a button on the screen.

Once the animation is started, the scheduling thread will maintain a list of the flights currently airborne. This is stored in `ActiveFlights`. Information about a flight is stored in a `FlightElement` object. When a traffic snapshot has been calculated, it is stored in a `Traffic` object, which constains a `TrafficElement` object for each flight active in that snapshot. The animation thread draws the aircraft by using the information in a `TrafficElement` object.

3.8 Alternative Projections

As mentioned in section 3.2 on page 15, the ATS system is supplied with two different projections. Although these should suffice for most applications, the system has been designed to allow other projections to be added.

A new projection is created by writing a Java class that extends the class `ats.airspace.Projection` and provides implementations for the abstract methods.

The public interface of `ats.airspace.Projection` is listed below.

```
public abstract class Projection {
  public abstract void drawPoint(Point2 p);
  public abstract void drawPoint(Point3 p);
  public abstract void drawLine(Point2 p1, Point2 p2);
  public abstract void drawLine(Point3 p1, Point3 p2);
  public abstract void drawGeodesic(Point2 p1, Point2 p2);
  public abstract void drawGeodesic(Point3 p1, Point3 p2);
  public abstract void drawCircle(Point2 p, Point1 radius);
  public abstract void drawPolygon(Polygon p);
  public abstract void drawString(String s, Point2 p);
  public abstract void drawString(String s, Point2 p, int line);
  public abstract Point2 getCoordsFor(java.awt.Point p);
  public abstract void setCenter(Point2 p);
  public abstract void zoom(Point2 p, float factor);
  public abstract void setBounds(Point3 p1, Point3 p2);
  public void keepAspectRatio(boolean b);
  public Graphics getGraphics();
  public void setGraphics(Graphics gr, Dimension d);
  public void clear();
  public void setColor(Color c);
  public void setBackground(Color c);
  public abstract String getType();
  public abstract String getDescription();
  public abstract String getProperties();
}
```

Note that while the two projections supplied with the ATS system are similar to those used for drawing a map, there are other possibilities – for example a three dimensional projection or a projection using parallel coordinates as described in [5]. The interface of `Projection` should be general enough to accomodate such variations.

For more information about projections and cartography in general refer to [2].

Chapter 4

Simulation Scenarios

In this section we present two simulation scenarios that demonstrate the functionality of the ATS system.

4.1 Case I: Billund Airport

The first scenario looks at the traffic to and from Billund Airport on the 26th April 2000[1]. During that day there were 22 scheduled departures and 34 scheduled arrivals. The first aircraft of the day arrived at 9.15 am and the last at 10.50 pm. There were services to 18 European airports. The time table used is shown in tables 4.1 and 4.2.

The scenario demonstrates the primary features of the simulator on a small example.

Figure 4.1 shows the time table as it is displayed in the program. Flights are sorted in ascending order of takeoff time. Figure 4.2 show the same time table, but now represented as an airport time table. This representation focuses on the airports rather than the flights. A flight will appear under both departure and destination airport.

Figure 4.3 shows a screen snapshot taken during the simulation of the scenario. Billund airport (with airport code BLL) is near the centre of the image. The yellow curves originating from Billund are airways to London, Paris, Las Palmas, Riga and Oslo. The short lines are aircraft indicators. These extend ahead of the aircraft in the direction the aircraft is heading. Their length is proportional to the horizontal speed of the aircraft. Shown below the aircraft indicators are the flight identification number and the altitude of the aircraft in 100 ft units.

In the snapshot there are five aircraft heading for Billund. Note how the aircraft BA8266 is following the airway between Oslo (OSL) and Billund.

Figure 4.4 shows the screen displayed once the simulation is finished. At the top of the image is shown the time (in the simulated world) when the

[1] Traffic listings are taken from the Billund Airport homepage at `http://www.bll.dk`.

	Time	*Flight number*	*Destination*
Departures	9.35	DM354	Copenhagen
	11.20	LH9357	Frankfurt
	11.50	SK8441	Amsterdam
	12.00	DM247	Dublin
	12.15	DM042	Aalborg
	12.20	DM205	London Gatwick
	13.50	BA8247	Manchester
	15.15	SK8478	Copenhagen
	16.40	BA8275	Berlin Tegel
	16.40	BU506	Oslo
	16.55	DM257	Brussels
	17.00	AF8003	Paris Charles de Gaulle
	17.05	LH9359	Frankfurt
	17.15	DM279	Amsterdam
	17.20	DM209	London Gatwick
	17.25	T4476	Riga / Jönköping
	17.30	SK8425	Stockholm
	17.50	BA8267	Oslo
	18.15	SK8480	Copenhagen
	18.45	TT423	Kaunas / Palanga
	19.30	DM366	Copenhagen
	21.50	DM046	Aalborg

Table 4.1: Scheduled departures from Billund Airport on 26th April 2000.

Arrivals

Time	*Flight number*	*Departure airport*
9.15	DM353	Copenhagen
10.40	SK8424	Stockholm
10.40	BA8266	Oslo
10.45	DM272	Amsterdam
11.00	BA8274	Berlin Tegel
11.10	SN5039	Brussels
11.15	LH9354	Frankfurt
11.35	BA8246	Manchester
11.40	DM202	London Gatwick
11.45	AF8000	Paris Charles de Gaulle
14.55	SK8477	Copenhagen
15.30	LH9356	Frankfurt
15.35	DM274	Amsterdam
16.05	BU505	Oslo
16.05	SK8446	Dublin
16.25	DM206	London Gatwick
16.30	DM045	Aalborg
16.45	T4475	Riga / Jönköping
17.10	NB472	Las Palmas
17.40	DM363	Copenhagen
18.00	TT422	Kaunas / Palanga
19.10	SK8487	Copenhagen
19.40	BA8268	Gothenburg
20.05	BA8276	Berlin Tegel
20.35	DM228	Stockholm
20.35	DM280	Amsterdam
20.50	DM258	Brussels
21.05	BA8268	Oslo
21.15	DK688	Las Palmas
21.15	DM210	London Gatwick
21.15	LH9358	Frankfurt
21.20	AF8002	Paris Charles de Gaulle
21.50	BA8248	Manchester
22.50	DM369	Copenhagen

Table 4.2: Scheduled arrivals at Billund Airport on 26th April 2000.

simulation was started and stopped. Just below there is a graph showing how many aircraft took off and landed at Billund airport accumulated over 15 minute intervals. The first axis is time and the second axis is the accumulated number of aircraft. At the bottom it reads that 15 spacing violations were detected during the simulation. A spacing violation occurs when two aircraft are closer than regulations permit. The regulations used for this simulation require a horizontal distance of at least 6 nm and a vertical distance of at least 1000 ft.

Flight number	Airport	Arrival time	Departure time
DM353	Kastrup (CPH)		Wed Apr 26 08:49 2000
	Billund (BLL)	Wed Apr 26 09:15 2000	
DM354	Billund (BLL)		Wed Apr 26 09:35 2000
	Kastrup (CPH)	Wed Apr 26 10:01 2000	
DM272	Schiphol (AMS)		Wed Apr 26 09:47 2000
	Billund (BLL)	Wed Apr 26 10:45 2000	
SK8424	Stockholm Arlanda (ARN)		Wed Apr 26 09:54 2000
	Billund (BLL)	Wed Apr 26 10:40 2000	
SN5039	Brussels (BRU)		Wed Apr 26 09:55 2000
	Billund (BLL)	Wed Apr 26 11:10 2000	
BA8266	Oslo Gardermoen (OSL)		Wed Apr 26 10:03 2000
	Billund (BLL)	Wed Apr 26 10:40 2000	
BA8274	Berlin Tegel (TXL)		Wed Apr 26 10:06 2000
	Billund (BLL)	Wed Apr 26 11:00 2000	
LH9354	Frankfurt (FRA)		Wed Apr 26 10:38 2000
	Billund (BLL)	Wed Apr 26 11:15 2000	
BA8246	Manchester (MAN)		Wed Apr 26 10:44 2000

Figure 4.1: Screen snapshot from scenario 1.

4.2 Case II: Random Time Table

The second scenario is based on a random time table. It includes all the airports of the first scenario and adds a further 17 airports in Europe, North America and Asia. The time table features a total of 500 flights on 3th May 2000 with the first departure at 5.00 am and the last takeoff at 16.30 pm. Aircraft are allowed to fly between any two airports.

The second scenario demonstrates the ability to handle larger simulations involving many airports and many flights. It also shows how the system may be used to locate areas where the concentration of aircraft is too great and where there is spare capacity.

Figure 4.5 covers most of central Europe and gives an impression of the amount of air traffic. It also shows two aircraft that are violating the spacing

ATS Editor

File

Airport	Flight number	Arrival time	Departure time
Aalborg	DM046	Wed Apr 26 22:09 2000	
	DM045		Wed Apr 26 16:11 2000
	DM042	Wed Apr 26 12:34 2000	
Berlin Tegel	BA8276		Wed Apr 26 19:13 2000
	BA8275	Wed Apr 26 17:32 2000	
	BA8274		Wed Apr 26 10:06 2000
Billund	NB472	Wed Apr 26 17:10 2000	
	TT423		Wed Apr 26 18:45 2000
	TT422	Wed Apr 26 18:00 2000	
	BA8284	Wed Apr 26 19:40 2000	
	DM279		Wed Apr 26 17:15 2000
	DM274	Wed Apr 26 15:35 2000	
	DM272	Wed Apr 26 10:45 2000	
	DM247		Wed Apr 26 12:00 2000
	SK8487	Wed Apr 26 19:10 2000	
	SK8480		Wed Apr 26 18:15 2000
	DM046		Wed Apr 26 21:50 2000

Figure 4.2: Screen snapshot from scenario 1.

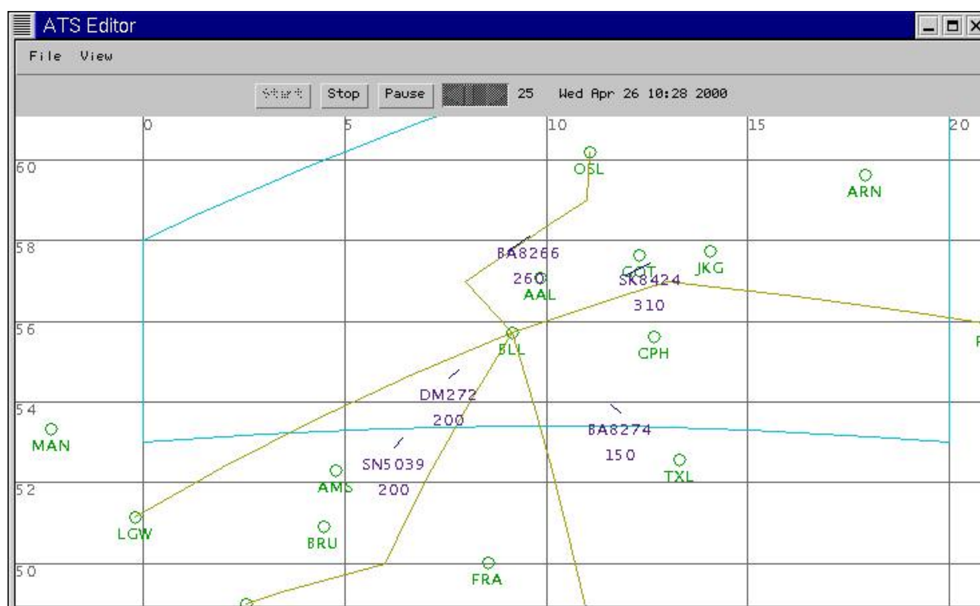

Figure 4.3: Screen snapshot from scenario 1.

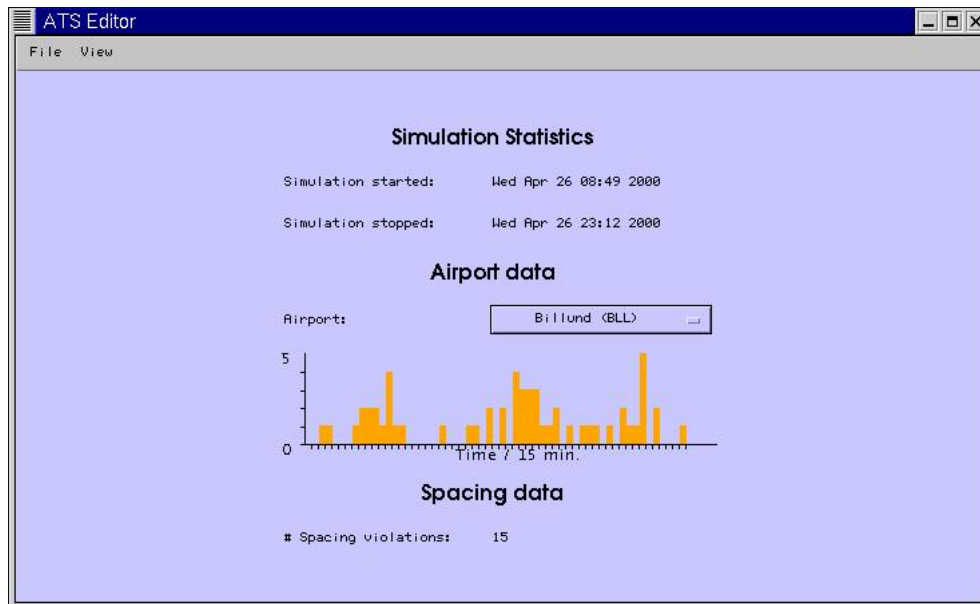

Figure 4.4: Screen snapshot from scenario 1.

regulations. These aircraft have their flight numbers drawn in red on the screen. Also note that in this case the projection has been changed to the `ConformalConicalProjection` as opposed to the `CylindricalProjection` used in the first scenario.

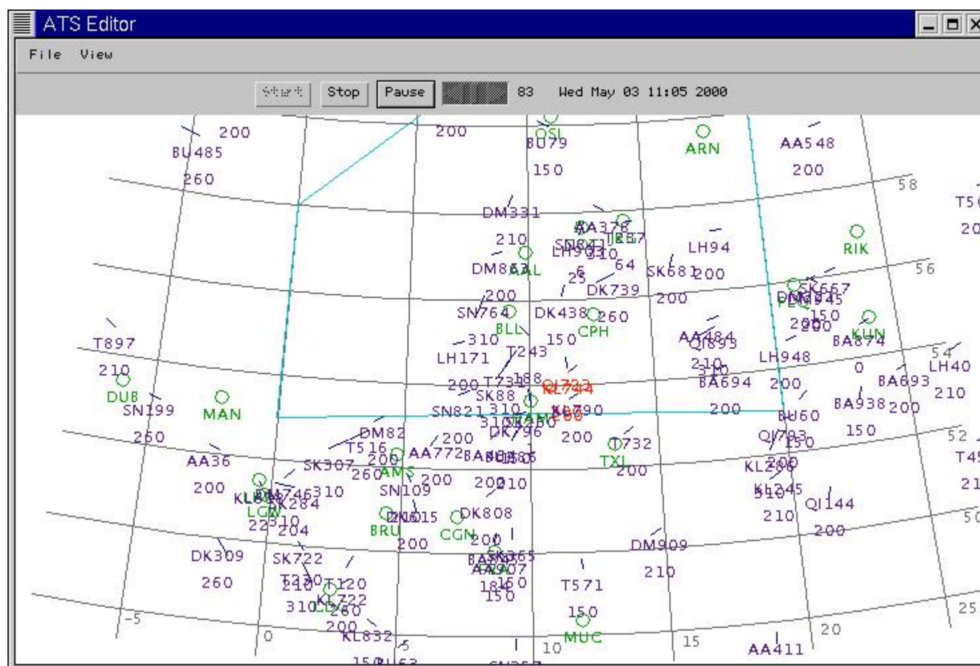

Figure 4.5: Screen snapshot from scenario 2.

Chapter 5

Conclusions

We have developed an Air Traffic Simulation system in Java capable of modelling air traffic, and presenting it through static and dynamic graphics.

Some of the important basic data structures we have completed concern the airspace and time table. Furthermore, a graphical user interface has been provided, integrating the basic data structures and their functions. This interface also allows users to save their work and regret changes made to the airspace or time table. Moreover, we have developed an animation module that utilizes the time table and airspace data structures.

The system is a useful tool for planning flight schedules because the animation will reveal any problems in the schedule. We have demonstrated the program's ability to simulate schedules including as many as 500 flights with no problems. For larger schedules the program needs some optimizations to work smoothly.

The data structures are capable of being upgraded to simulate the real world more accurately, as well as add further functionality to our program such as predicting near misses, and automatically resolving such conflicts. The program can also be equipped with other, more advanced, projections. This extension is relatively easy because of the airspace interface.

Appendix A

User's Guide

This document is a brief user's guide on the Air Traffic Simulation application. For the sake of simplicity, we will refer to the application as ATS.

A.1 Starting the Program

The ATS program was developed with Java 2, and will therefore require the presence of a Java 2 installation on the machine the program is to run on. Refer to your Java documentation on how to set up the Java software, or download it from `java.sun.com`.

All files related to the ATS program can be downloaded from

```
www.student.dtu.dk/~c973746/ats/
```

The file you need to download is ats.zip. Extract this file in a new directory. This is done by executing the command in your new directory (from the console window for UNIX users or at the DOS prompt for Windows users):

```
jar xf ats.zip
```

To start the application, you must place yourself in the directory where you extracted the ZIP file. Once there, you start the program by entering the following command:

```
java ats.gui.WelcomeFrame
```

The application will then begin, and the first window will appear: The WelcomeFrame.

A.2 The WelcomeFrame and Getting Started

The WelcomeFrame is the starting point of every user session. Here you are presented with the title screen of ATS, and the authors names.

A.2.1 The File Menu Bar

At the top of the screen, you find the Menu Bar. In the WelcomeFrame, there is only one Menu to choose from, and that is File. Its functions, listed in order from top to bottom, are as follows:

- New Session takes you into the program, allowing you to start a Session from scratch by defining or loading an Airspace as well as choosing an empty or saved Time Table. Once done, you are taken to the EditorFrame where all aspects of ATS become accessible.
- Open Session allows you to select a saved Session file, load it and to continue to work with it.
- About takes you to a new screen where some supplementary information concerning ATS is found. To return to the WelcomeFrame, press the OK button at the bottom of the window.
- Exit quits the ATS application, and saves whatever changes were made to the properties of ATS during the your last session. It is highly recommended that this be the only way that you terminate any Session.

A.2.2 New Session

When starting a new session, you can either choose to load an Airspace from disk, or to create an Airspace from scratch. All Airspace files have the extension `.asp`. However, if you choose to create an Airspace from scratch you are presented with a screen where you are asked to define this new Airspace.

You must first define how many corners you want in your Airspace, since an Airspace shape is that of a polygon. Also needed are the maximum and minimum altitudes of the Airspace.

When you have entered the data and continue, you are presented sequentially with screens, each one asking you to input data for the corners of this polygon. The data needed are the latitude and longitude of the corners. The origin is placed where the equator crosses the Greenwich longtitude. A positive number will take you either north or east, while a negative number will take you south or west. For example, a latitude of -5 degrees and a longtitude of 20 degrees will take you to 5 degrees south, and 20 degrees east. Keep in mind that if you wish to change the shape of the Airspace, you will have to restart ATS and create a new session from scratch.

Once you have decided on the Airspace for your new session, you are taken to a screen where you must decide how your TimeTable should be. To begin with, you have only two options to choose from. You may either start the session with an empty time table, or load one from disk. All TimeTable files have the extension `.ttb`.

Keep in mind, that you may only use a TimeTable whose flights fly to and from airports that exist in your chosen Airspace. If this is not the case, an error will notify you of this problem. Once you have chosen a TimeTable, you are taken to the EditorFrame part of ATS, where all aspects of ATS are accessible.

A.2.3 Open Session

If you have worked with ATS before, and saved a Session file (extension `.ses`), you may continue your work by choosing the Open Session command from the File menu. This opens a dialog box where you may browse for your file, and load it.

When you have selected a file, and loaded it, you are taken to the very beginning of your saved session, and not the end! If you wish to work from where you left off, you must move forward through your session by pressing Ctrl + S.

A.3 The EditorFrame

This is the core of the ATS program, where you work with your selected Airspace and TimeTable. The menu bar consists only of the File menu and Edit menu, and these are also displayed at all times when using the many utilities in ATS.

A.3.1 The File Menu Bar

The File Menu bar allows you to access the various portions of the ATS program, and provides access to altering some properties of the the whole application. It is also from this menu that you can restart ATS, and return to the WelcomeFrame, whereby you can exit the program completely if that is your intention.

In order, from top to bottom, the File Menu consists of:

- Time Table takes you to the TimeTable module of ATS, where you can create, modify or remove Aircrafts and Flights.
- Airspace takes you to the Airspace module of ATS, where you can view a static representation of the Airspace. Here, you can also add, modify or remove Airports and Airways.

- Animation takes you to the Animation module of ATS where you can view an accelerated simulation of how all the flights in the Time Table make it in the Airspace. A statistical resume ends this module.

- Properties consists of two sections. The Time Table and Airspace module properties. By selecting the Time Table properties, you can set how much time each flight should wait between landing at an airport, and taking off again. This change does not affect previously scheduled flights. Selecting the Airspace properties, you can alter how the Airspace is visually displayed.

- Save Session allows you to save your work into a Session file. By doing this, you can quit the program, and then return to your work in progress by loading the saved file.

- Restart ATS returns you to the WelcomeFrame screen, and you are prompted if you want to save your session in progress. If you do not save, your work is discarded.

A.3.2 The Edit Menu Bar and the Undo/Redo Functionality

This menu controls a useful aspect of the ATS program. When you work, the program detects important changes to the Airspace and TimeTable you are dealing with, and saves it. This allows you to move back and forth in your work, and see how you've come to where you are. More importantly, it allows you to move back if you have regretted a step you've done before.

What is important to note is that you can freely move back and forth throughout the entire history of your session, but if you move back and cause an important change to your TimeTable or Airspace, you discard any saved history ahead of your position when you effectuated the change.

- Previous takes you one step back in your session, and can be performed by pressing Ctrl + Z.

- Next takes you one step forward in your session, and can be performed by pressing Ctrl + A.

- Start takes you to the beginning of your session, and can be performed by pressing Ctrl + X.

- End takes you to the end of your session, and can be performed by pressing Ctrl + S.

Another important note is that when you load a saved session, you are presented with the first important event you performed in your saved session. If you wish to continue from where you left off and keep the history of your session intact, you must either press Ctrl + S or select End from the Edit menu.

A.4 The Airspace Module

The Airspace Module is where you can view a static representation of the Airspace. Here, you can also add, modify or remove Airports and Airways.

A.4.1 AS Options Menu

The Airspace module has supplementary additions to the standard menu bar of the EditorFrame. The first is the AS Options menu.

- Airport (Insert) allows you to insert an airport in the airspace.
- Airport (Remove) will give you a list over airports, which you can remove.
- Airport (Modify) gives you a list over airports, and lets you modify their properties.
- Airway (Insert) allows you to create an airway between two airports.
- Airway (Remove) allows you to remove an existing airway between two airports.
- Save Airspace allows you to save your actual airspace into a loadable `.asp` file.

A.4.2 View Menu

This menu allows you to move about in the airspace with your mouse. To do this, you select which command you want in the menu, and then click on the airspace where your interest is.

Another feature of the airspace visualization is that by right clicking an airport, a tiny pop up box will appear, detailing some information about the airport.

- Zoom In causes the view of the airspace to zoom in where the mouse is clicked.
- Zoom Out expands the view, centering it on where you clicked in the airspace.
- Center centers the view of the airspace on where you clicked.

A.4.3 Airports

When operating with airports in the Airspace module, you have 3 options open to you, which will be dealt with here in order. These options are Insert, Remove and Modify.

When you select Insert in the submenu of Airport under the AS Options menu, the screen is split in two, where on the right half of the screen you have a view of the airspace, and on the left hand of the screen you have the area where you define the airport to be inserted.

On this left area of the window you have to specify the Airport's name, it's 3 letter ID code, its position on Earth in degrees longitude and latitude, and its height above sea level. When this is done, the airport will be inserted into the airspace, and will be reflected accordingly on the visualization of the airspace on the right side of the window.

The left area of the screen will remain, allowing you to enter a new airport without having to access the command through the menubar yet again.

When removing an airport, make sure that no flights are scheduled to arrive or depart from the airport. When the airport is not present in the TimeTable at all, removing it becomes a simple process. When the option is selected form the AS Options menu, the screen splits up again in a similar fashion as for the Insert command. On the left side of the window, a list of airports will be available in a choice box, and you select which one to be removed. Once selected, removing the airport is a matter of pressing the remove button.

To modify an airport, as in the case of removing an airport, you must make sure that no flights are scheduled to arrive or depart from the airport. Modifying an airport splits the screen again, where you have a choice of available airports on the left side of the window. You select which airport you wish to modify, and press the modify button. Once this is done, you can alter the old values of the airport, and when satisfied, you press the Replace button.

A.4.4 Airways

Airways serve to guide flights between two airports along a set route in the sky, and this part of the program is designed to help you create them in the Airspace.

Airways can either be inserted or removed, and if you wish to modify an airway, you would have to remove it first, and insert a new one in its place.

When you choose to insert an airway, the window splits again, and on the left side of the screen, you are prompted to select the two airports to connect, the capacity of the airway, it's width, and how many straight segments it is to be composed of.

When this is defined, the next series of screens (one for each segment)

allows you to choose the segments ending coordinate (unless only 1 segment has been chosen for the airway), its minimum altitude and maximum altitude. When this is done for each segment, it is finally inserted into the airspace.

Removing an airway is done in a similar fashion, although you only have to select which two airports the specific airway connects. Once the two airports have been chosen, pressing the remove button will erase the airway connection between the two airports.

A.5 The Time Table Module

The Time Table Module is the area of ATS where you can define what types of aircraft exist, modify existing aircraft, as well as create, schedule and otherwise manipulate flights.

A.5.1 TT options menu

- Flight Insert allows you to insert a new flight, specifying its destinations and schedule.
- Flight Remove lets you erase a flight from the time table.
- Aircraft Create lets you create an aircraft type. Without aircraft, you can't have flights.
- Aircraft Remove lets you remove a type of aircraft.
- Aircraft Modify allows you to modify the properties of existing aircraft.
- Landing Insert allows you to insert a destination in the schedule of an existing flight.
- Landing Remove lets you erase a stop in an existing flight.
- Save Time Table permits you to save your time table to a loadable `.ttb` file.
- View Airport Time Table sorts the time table according to airports, instead of flights.
- Print to File sends the entire data of the time table into text in a user specified file.

A.5.2 Aircraft

Before you can actually go ahead and create flights, you must specify at least what sort of aircraft these flights use. To do this, you can use the Flight Insert selection in the menu bar, which brings you to a screen where you can name your aircraft, and assign it many different sorts of values defining its characteristics.

To remove an aircraft, you select the Aircraft Remove option. You are then presented with a list of aircraft names, and to remove one, you simply select it from the list, and press the remove button. This will not be possible, however, if an aircraft is currently servicing any flights.

A.5.3 Flights

Once you have aircraft, you need to specify where they fly and when. Creating a flight is straightforward. Firstly, you need to select Aircraft Create from the menu bar. Once done, you are presented with the task of choosing a flight number (such as SK3003 for SAS flight 3003), a list of aircraft types to choose from for this flight, as well as how many intermediate landings are to be made between the origin airport and end airport.

Then for each point on the journey you are presented with a screen where you must select what that point is and when you depart that point. The airports you can choose from are those which lie within range of the aircraft, and after the first point on the journey is defined, the arrival time for the subsequent points is displayed above the airport choice box. This helps you in selecting a proper departure time. The last point on the journey requests the user of only one piece of information: what that destination is.

When completed, the newly created flight will be added to time table.

A.5.4 Landings

Once flights have been defined, and you wish to add or remove any intermediate landings in them, you can do this through Landings under the TT options menu.

Choosing Landings Insert lets you choose what flight number should have a new landing, as well as what airport that landing is at, and when the flight departs the airport. The landing is then inserted into its proper area in the flights schedule.

Choosing Landings Remove will allow you to remove a landing from a flight provided that it has 3 or more points on its journey. Here, you only have to specify what flight number must have a landing removed, which airport destination is to be removed, and if the flight lands multiple times at this airport, the user selects which departure time the landing has from the airport in order to get the right landing removed.

A.5.5 Other Options

Other options available to you is to view the Time Table according to airports instead of according to flights. This presents you with a table where each airport appears once, followed with its long list of flights and their data. This reminds one much of the arrivals/departures screens located in an airport.

Another option is to print the Time Table data out into file in a textual representation. To do this, you simply select Print to File from the TT options menu, and specify what the file should be called.

A.5.6 The Table

The actual Time Table itself is dynamic in behaviour, allowing the user to adjust it without operating through the menu commands. The arrival and departure times are automatically updated, whenever the user takes action which alters the characteristics of a flight.

The aircraft a flight has can be viewed by clicking on the flight number in the far left column of the time table. A popup window appear detailing the properties of the assigned aircraft. The user can then choose to rename the flight or choose a different aircraft to service the flight here.

Clicking on any airport name in the second column provides a drop down list by which the user can select a new airport that the flight should land at instead.

While the Arrival Times column is immutable, since their values are calculated directly whenever changes are made, the only way a user can modify the scheduling of a flight with respect to time, is to alter departure times. This can be done, so long as the user doesn't enter a time which causes the plane to take off too early, as set by the minimum time separation variable in the Time Table properties menu point.

While the Table sorted according to Flights is readily editable in this fashion, the Table sorted according to Airports is not.

A.6 The Animation Module

This area of the program takes the scheduled flights from the TimeTable module of the program, and combines it with the Airspace, allowing you to view how your flights travel according to their schedules.

Selecting it will take you to the animation, where at first it will be paused. The date and time can be read off near the top, and where you will also find other controls over the animation.

Pressing start will commence the animation and automatically begin with the earliest scheduled flight in the TimeTable module. To pause and unpause, simply press the pause button. A slider is also available allowing you to adjust the speed at which time flows.

When you finally decide to end the animation, you are taken to a screen where you are presented with an overview of statistics concerning the time span you viewed. From here, you can move on in the ATS program.

A.7 The Properties Menu

This area of ATS allows you to control certain aspects of the program for visualizing the Airspace or how long a plane should wait between landings and take offs.

A.7.1 Time Table Properties

The Time Table properties area only allows you to change how many minutes a plane must be on the ground before it is allowed to take off after it has landed. Changing this value does not affect already scheduled flights, but once it has been changed, modyfing the existing flights or creating new ones will make scheduling behave accordingly to the new setting.

A.7.2 Airspace Properties

The Airspace properties panel is mainly concerned with the visual properties when displaying the Airspace. You can select different colors for representing Airports, Airways, the Airspace boundary, and the background (Airspace Canvas) as well as what their highlighted colors should be.

You can also select whether or not the longitude/latitude grid lines should be visible, or whether the aspect ratio should be kept when zooming in or out.

Another important feature for visualing the airspace is to choose which projections should be available. These are made available as a drop-down list, and choosing one will cause a description of the projection to appear in the bottom right area of the window.

At the time of release, two projections were made available: Cylindrical and Conformal Conical projections.

A.8 Contact Information

Should questions or comments arise, you may contact the development team for help. Fan mail is also welcomed. `;-)`

- Graphical User Interface and Undo/Redo Functionality :
  Mark Hoffmann
  c973500@student.dtu.dk
  Dan Erik Petersen
  c973539@student.dtu.dk

- Airspace and Animation :
  Christian Krog Madsen
  c973746@student.dtu.dk
- TimeTable :
  Kristine Frank
  c973732@student.dtu.dk

Appendix B

System Verification

In this chapter we give examples of how the software system has been tested and verified.

B.1 Datastructure Tests

The part of the ATS system that stores time tables, demo sequences and air spaces has been tested method by method.

We illustrate how the testing was done with the class `ats.airspace.Name`. The tests for the remaining data structure classes are listed in the HTML files `AirspaceTestResult.html`, `AnimationTestResult.html`, `TimeTableTestResult.html` and `DemoTestResult.html` available from the ATS homepage. See appendix A for details.

Table B.1 lists the test cases identified. The source code for the class is listed below. For each test case the method is called with the arguments listed in the table, and the returned value is compared with the expected result. Note that in some cases "expected value" refers to a change in the internal data structure. In these cases the internal structure is inspected and compared with the expected structure.

The notation used is table B.1 is as follows. Objects are written as their class name followed by the values of internal fields enclosed in square brackets. The contents of arrays, vectors and hashtables are shown in curly braces. Values may be assigned an alias, i.e. `w1 = Width[5]` means that in the following `w1` refers to a width of 5 units.

```
package ats.airspace;

public class Name implements java.io.Serializable, java.lang.Cloneable {

  static final long serialVersionUID = -8990558932145611793L;

  String n = null;
  String i = null;
```

Test	*Object*	*Parameters*	*Expected value*
Name(String, String)			
1a	–	“Kastrup”, “CPH”	n1 = Name[“Kastrup”, “CPH”]
1b	–	null, “CPH”	NullPointerException
1c	–	“Kastrup”, null	NullPointerException
Name(String)			
2a	–	“Aalborg (AAL)”	n2 = Name[“Aalborg”, “AAL”]
2b	–	“Aalborg”	NullPointerException
2c	–	“ ”	NullPointerException
getID()			
3a	n1	–	“CPH”
getName()			
4a	n1	–	“Kastrup”
clone()			
5a	n1	–	n3 = Name[“Kastrup”, “CPH”]
equals(Object)			
6a	n1	n3	true
6b	n1	n2	false
toString()			
7a	n1	–	“Kastrup (CPH)”
hashCode()			
8a	n1	–	-205369134
8b	n2	–	1426601704

Table B.1: Test cases for the class `ats.airspace.Name`.

```
public Name(String nameandid) throws NullPointerException {
  int ps = nameandid.indexOf('(');
  int pe = nameandid.indexOf(')');
  if(ps==-1 || pe==-1 || pe<ps)
    throw new NullPointerException("ID not found");
  String name = nameandid.substring(0,ps-1);
  String id = nameandid.substring(ps+1,pe);
  n = name;
  i = id.toUpperCase();
}

public Name(String name, String id) throws NullPointerException {
  if(name == null || id == null)
    throw new NullPointerException("Airport Name or ID null.");
  n = name; i = id.toUpperCase();
}

public String getName() {
  return n;
}
```

```
  public String getID() {
    return i;
  }

  public String toString() {
    return n+" ("+i+")";
  }

  public int hashCode() {
    return n.hashCode() * i.hashCode();
  }

  public boolean equals(Object o) {
    return (o instanceof Name) && (((Name) o).i.equals(i)) && (((Name)
    o).n.equals(n));
  }

  public Object clone() {
    return new Name(new String(n), new String(i));
  }
}
```

B.2 GUI Test

The graphical user interface is tested through the ATS program at run-time. The menus are implicitly tested while using the program and therefore not a part of the following test. The GUI test is divided in two parts. Firstly all panels are tested by using the program in some ordinary as well as unusual cases. Secondly the getAppropiatePanel method from the ats.gui.EditorFrame class is tested.

B.2.1 Panel test

The panel test consist of two tables. The first table shows what there has been tested and the second table shows the test results. In the "Expected Result" column in the second table any response from the ATS program is denoted. However, when a method from one of the datastructures is invoked and all arguments are as expected there is no messages shown from the ATS program. In this case it is checked that the right methed is invoked which is denoted by "Method call: METHOD". This is checked by adding a simple "print to screen" line in the source code just before a method is invoked. Some conventions are made: 1. *Any*Panel[xxx] means that *Any*Panel is set up for xxx environments. 2. Values in a panel are listed by [x,y,z] in that same order as the panel is filled.

Case	Frame	Panel	Test
1a	WelcomeFrame	SelectPanel[Airspace Selection]	OK button. "New Airspace" is selected.
1b	WelcomeFrame	SelectPanel[Airspace Selection]	OK button. "Open Airspace" is selected.
2a	WelcomeFrame	SelectPanel[Define Airspace]	OK button. Number of corners in polygon isn't an integer.
2b	WelcomeFrame	SelectPanel[Define Airspace]	OK button. Number of corners in polygon is less than three.
2c	WelcomeFrame	SelectPanel[Define Airspace]	OK button. Minimum altitude isn't a floating number.
2d	WelcomeFrame	SelectPanel[Define Airspace]	OK button. Maximum altitude isn't a floating number.
2e	WelcomeFrame	SelectPanel[Define Airspace]	Reset button.
2f	WelcomeFrame	SelectPanel[Define Airspace]	OK button. Arguments OK.
3a	WelcomeFrame	SelectPanel[Define Point]	OK button. Longitude isn't a floating number.
3b	WelcomeFrame	SelectPanel[Define Point]	OK button. Latitude isn't a floating number.
3c	WelcomeFrame	SelectPanel[Define Point]	Reset button.
3d	WelcomeFrame	SelectPanel[Define Point]	OK button. Arguments OK.
4a	WelcomeFrame	SelectPanel[TimeTable Selection]	OK button. "New Time Table" is selected.
4b	WelcomeFrame	SelectPanel[TimeTable Selection]	OK button. "Open Time Table" is selected.
5a	EditorFrame	ASEditAirportPanel	Insert button. No Airport Name.
5b	EditorFrame	ASEditAirportPanel	Insert button. Airport ID not a three letter word.
5c	EditorFrame	ASEditAirportPanel	Insert button. Longitude isn't a floating number.
5d	EditorFrame	ASEditAirportPanel[Insert]	Insert button. Latitude isn't a floating number.
5e	EditorFrame	ASEditAirportPanel[Insert]	Insert button. Elevation isn't a floating number.

5f	EditorFrame	ASEditAirportPanel[Insert]	Insert button. Longitude out of bounds.
5g	EditorFrame	ASEditAirportPanel[Insert]	Insert button. Latitude out of bounds.
5h	EditorFrame	ASEditAirportPanel[Insert]	Reset button.
5i	EditorFrame	ASEditAirportPanel[Insert]	Insert button. Arguments OK.
6a	EditorFrame	ASEditAirportPanel[Remove]	Remove button. Drop down box is empty.
6b	EditorFrame	ASEditAirportPanel[Remove]	Remove button. Removing an airport that exist in the time table.
6c	EditorFrame	ASEditAirportPanel[Remove]	Remove button. Arguments OK.
7a	EditorFrame	ASEditAirportPanel[Modify]	Modify button. Drop down box is empty.
7b	EditorFrame	ASEditAirportPanel[Modify]	Modify button. Modifying an airport that exist in the time table.
7c	EditorFrame	ASEditAirportPanel[Modify]	Modify button. Arguments OK.
8a	EditorFrame	ASEditAirportPanel[Replace]	Replace button. Longitude isn't a floating number.
8b	EditorFrame	ASEditAirportPanel[Replace]	Replace button. Latitude isn't a floating number.
8c	EditorFrame	ASEditAirportPanel[Replace]	Replace button. Elevation isn't a floating number.
8d	EditorFrame	ASEditAirportPanel[Replace]	Replace button. Longitude out of bounds.
8e	EditorFrame	ASEditAirportPanel[Replace]	Replace button. Latitude out of bounds.
8f	EditorFrame	ASEditAirportPanel[Replace]	Reset button.
8g	EditorFrame	ASEditAirportPanel[Replace]	Replace button. Arguments OK.
9a	EditorFrame	ASEditAirwayPanel[Insert]	Next button. Capacity isn't an integer.
9b	EditorFrame	ASEditAirwayPanel[Insert]	Next button. Width isn't a floating number.
9c	EditorFrame	ASEditAirwayPanel[Insert]	Next button. Number of segments isn't an integer.

9d	EditorFrame	ASEditAirwayPanel[Insert]	Next button. Number of segments is less than one.
9e	EditorFrame	ASEditAirwayPanel[Insert]	Next button. Airway between one airport.
9f	EditorFrame	ASEditAirwayPanel[Insert]	Reset button.
9g	EditorFrame	ASEditAirwayPanel[Insert]	Next button. Arguments OK.
10a	EditorFrame	ASEditAirwayPanel[Define Segment(1 of 2)]	Next button. End longitude isn't a floating number.
10b	EditorFrame	ASEditAirwayPanel[Define Segment(1 of 2)]	Next button. End latitude isn't a floating number.
10c	EditorFrame	ASEditAirwayPanel[Define Segment(1 of 2)]	Next button. Minimum altitude isn't a floating number.
10d	EditorFrame	ASEditAirwayPanel[Define Segment(1 of 2)]	Next button. Maximum altitude isn't a floating number.
10e	EditorFrame	ASEditAirwayPanel[Define Segment(1 of 2)]	Reset button.
10f	EditorFrame	ASEditAirwayPanel[Define Segment(1 of 2)]	Next button. Arguments OK.
11a	EditorFrame	ASEditAirwayPanel[Define Segment(2 of 2)]	Reset button.
11b	EditorFrame	ASEditAirwayPanel[Define Segment(2 of 2)]	Next button. Arguments OK
12a	EditorFrame	ASEditAirwayPanel[Remove]	Remove button. Drop down boxes are empty.
12b	EditorFrame	ASEditAirwayPanel[Remove]	Remove button. Remove airway between one airport.
12c	EditorFrame	ASEditAirwayPanel[Remove]	Remove button. Remove airway between two airports where no airway is found.
12d	EditorFrame	ASEditAirwayPanel[Remove]	Remove button. Arguments OK.
13a	EditorFrame	TTEditAircraftPanel[Create]	Create button. No aircraft type.
13b	EditorFrame	TTEditAircraftPanel[Create]	Create button. Maximum speed isn't a floating number.

13c	EditorFrame	TTEditAircraftPanel[Create]	Create button. Cruise speed isn't a floating number.
13d	EditorFrame	TTEditAircraftPanel[Create]	Create button. Take off/landing speed isn't a floating number.
13e	EditorFrame	TTEditAircraftPanel[Create]	Create button. Maximum vertical speed isn't a floating number.
13f	EditorFrame	TTEditAircraftPanel[Create]	Create button. Range isn't a floating number.
13g	EditorFrame	TTEditAircraftPanel[Create]	Create button. Maximum turning rate isn't a floating number.
13h	EditorFrame	TTEditAircraftPanel[Create]	Create button. Maximum acceleration isn't a floating number.
13i	EditorFrame	TTEditAircraftPanel[Create]	Create button. Cruise altitude isn't a floating number.
13j	EditorFrame	TTEditAircraftPanel[Create]	Create button. Cruise speed is greater than max speed.
13k	EditorFrame	TTEditAircraftPanel[Create]	Reset button.
13l	EditorFrame	TTEditAircraftPanel[Create]	Create button. Arguments OK.
13m	EditorFrame	TTEditAircraftPanel[Create]	Create button. Create an existing aircraft.
14a	EditorFrame	TTEditAircraftPanel[Remove]	Remove button. Drop down box is empty.
14b	EditorFrame	TTEditAircraftPanel[Remove]	Remove button. Arguments OK.
15a	EditorFrame	TTEditAircraftPanel[Modify]	Modify button. Drop down box is empty.
15b	EditorFrame	TTEditAircraftPanel[Modify]	Modify button. Arguments OK.
16a	EditorFrame	TTEditAircraftPanel[Replace]	Reset button.
16b	EditorFrame	TTEditAircraftPanel[Replace]	Replace button. Arguments OK.
17a	EditorFrame	TTEditFlightPanel[Insert]	OK button. No flight number.
17b	EditorFrame	TTEditFlightPanel[Insert]	OK button. Aircraft drop down is empty.

17c	EditorFrame	TTEditFlightPanel[Insert]	OK button. Number of intermediate landings isn't an integer.
17d	EditorFrame	TTEditFlightPanel[Insert]	OK button. Number of intermediate landings is negative.
17e	EditorFrame	TTEditFlightPanel[Insert]	Reset button.
17f	EditorFrame	TTEditFlightPanel[Insert]	OK button. Arguments OK.
18a	EditorFrame	TTEditFlightPanel[Define Airport(1 of 2)]	OK button. Airport drop down box is empty.
18b	EditorFrame	TTEditFlightPanel[Define Airport(1 of 2)]	OK button. Departure day isn't an integer.
18c	EditorFrame	TTEditFlightPanel[Define Airport(1 of 2)]	OK button. Departure day is greater than 31.
18d	EditorFrame	TTEditFlightPanel[Define Airport(1 of 2)]	Reset button.
18e	EditorFrame	TTEditFlightPanel[Define Airport(1 of 2)	OK button. Arguments OK.
19a	EditorFrame	TTEditFlightPanel[Define Airport(2 of 2)]	OK button. Arguments OK.
20a	EditorFrame	TTEditFlightPanel[Remove]	Remove button. Drop down box is empty.
20b	EditorFrame	TTEditFlightPanel[Remove]	Remove button. Arguments OK.
21a	EditorFrame	TTEditLandingPanel[Insert]	OK button. Flight number drop down box is empty.
21b	EditorFrame	TTEditLandingPanel[Insert]	OK button. Departure time is wrong.
21c	EditorFrame	TTEditLandingPanel[Insert]	Reset button.
21d	EditorFrame	TTEditLandingPanel[Insert]	OK button. Arguments OK.
22a	EditorFrame	TTEditLandingPanel[Remove]	OK button. Arguments OK.
22b	EditorFrame	TTEditLandingPanel[Remove]	OK button. Remove landing when only two landing are present in the flight.
23a	EditorFrame	PropertiesPanel[TimeTable]	OK button. Separation time isn't a floating number.
23b	EditorFrame	PropertiesPanel[TimeTable]	OK button. Arguments OK.

Case	Values in Panel	Expected Result
1a	"New Airspace" is selected	SelectPanel[Define Airspace] is shown.
1b	"Open Airspace" is selected	Open FileChooser is shown.
2a	[xxx,0,20000]	warning("Invalid type of corners in polygon.")
2b	[2,0,20000]	warning("Number of corners in polygon must be at least three.")
2c	[4,xxx,20000]	warning("Invalid altitude min.")
2d	[4,0,xxx]	warning("Invalid altitude max.")
2e	[4,0,20000]	All textfields are empty.
2f	[4,0,20000]	SelectPanel[Define Points] is shown.
3a	[xxx,-1]	warning("Invalid longitude.")
3b	[-1,xxx]	warning("Invalid latitude.")
3c	[-1,-1]	All textfields are empty.
3d	[-1,-1]	SelectPanel[Define Points] is shown until all points are defined. The SelectPanel[TimeTable Selection] is shown when all point are defined.
4a	"New Time Table" is selected	EditorFrame is shown.
4b	"Open Time Table" is selected	Open FileChooser is shown.
5a	["",AP1,0,0,0]	warning("Invalid airport name.")
5b	[Airport1,APPP1,0,0,0]	warning("Invalid airport id. Must be a three-letter word.")
5c	[Airport1,AP1,xxx,0,0]	warning("Invalid longitude.")
5d	[Airport1,AP1,0,xxx,0]	warning("Invalid latitude.")
5e	[Airport1,AP1,0,0,xxx]	warning("Invalid elevation.")
5f	[Airport1,AP1,-300,0,0]	warning("Longitude out of bound-s! Longitude must between -180 and 180.")
5g	[Airport1,AP1,0,270,0]	warning("Latitude out of bounds! Latitude must be between -90 and 90.")
5h	[Airport1,AP1,0,0,0]	All textfields are empty.
5i	[Airport1,AP1,0,0,0]	Method call: createAirport.
6a	[""]	warning("Unknown airport")
6b	[AirportInTT (ATT)]	warning("Can't remove this airport. Check time table.")
6c	[Airport1 (AP1)]	Method call: removeAirport.
7a	[""]	warning("Unknown airport.")
7b	["AirportInTT (ATT)"]	warning("Can't modify this airport. Check time table.")
7c	[Airport1 (AP1)]	ASEditAirportPanel[Replace] is shown.

8a	[Airport1 (AP1),xxx,1,1]	warning("Invalid longitude.")
8b	[Airport1 (AP1),1,xxx,1]	warning("Invalid latitude.")
8c	[Airport1 (AP1),1,1,xxx]	warning("Invalid elevation.")
8d	[Airport1,AP1,-300,1,1]	warning("Longitude out of bound-s! Longitude must between -180 and 180.")
8e	[Airport1,AP1,1,270,1]	warning("Latitude out of bounds! Latitude must be between -90 and 90.")
8f	[Airport1 (AP1),1,1,1]	Longitude,latitude and elevation are empty, and Airport Name/ID isn't empty.
8g	[Airport1 (AP1),1,1,1]	Method: moveAirport
9a	[Airport1 (AP1),Airport2 (AP2),xxx,50,2]	warning("Invalid capacity.")
9b	[Airport1 (AP1),Airport2 (AP2),100,xxx,2]	warning("Invalid width.")
9c	[Airport1 (AP1),Airport2 (AP2),100,50,xxx]	warning("Invalid number of seg-ments.")
9d	[Airport1 (AP1),Airport2 (AP2),100,50,-1]	warning("Number of segments must be greater than zero.")
9e	[Airport1 (AP1),Airport1 (AP2),100,50,2]	warning("Invalid airports.")
9f	[Airport1 (AP1),Airport2 (AP2),100,50,2]	All textfields are empty
9g	[Airport1 (AP1),Airport2 (AP2),100,50,2]	ASEditAirwayPanel[Define Seg-ments(1 of 2)] is shown
10a	[xxx,0,3000,5000]	warning("Invalid end longitude.")
10b	[1,xxx,3000,5000]	warning("Invalid end latitude.")
10c	[1,0,xxx,5000]	warning("Invalid minimum alti-tude.")
10d	[1,0,3000,xxx]	warning("Invalid maximum alti-tude.")
10e	[1,0,3000,5000]	All textfields is empty except the s-tart long- and latitude textfields.
10f	[1,0,3000,5000]	ASEditAirwayPanel[Define Seg-ment(2 of 2)] is shown.
11a	[3000,5000]	Only min. and max. altitude textfields are empty.
11b	[3000,5000]	Method call: createAirway.
12a	["",""]	warning("Invalid airports.")
12b	[Airport1 (AP1),Airport1 (AP1)]	warning("Invalid airports.")
12c	[Airport1 (AP1),Airport3 (AP3)]	warning("No airways found.")
12d	[Airport1 (AP1),Airport2 (AP2)]	Method call: removeAirway.

13a	["",500,400,50,50,4000,50,50,1000]	warning("Invalid aircraft type.")
13b	[B52,"",400,50,50,4000,50,50,1000]	warning("Invalid maximum speed.")
13c	[B52,500,"",50,50,4000,50,50,1000]	warning("Invalid cruise speed.")
13d	[B52,500,400,"",50,4000,50,50,1000]	warning("Invalid take off and landing speed.")
13e	[B52,500,400,50,"",4000,50,50,1000]	warning("Invalid vertical speed.")
13f	[B52,500,400,50,50,"",50,50,1000]	warning("Invalid range.")
13g	[B52,500,400,50,50,4000,"",50,1000]	warning("Invalid maximum turn rate.")
13h	[B52,500,400,50,50,4000,50,"",1000]	warning("Invalid maximum acceleration.")
13i	[B52,500,400,50,50,4000,50,50,""]	warning("Invalid cruise altitude.")
13j	[B52,400,500,50,50,4000,50,50,""]	warning("Cruise speed can't be greater than max speed.")
13k	[B52,500,400,50,50,4000,50,50, 1000]	All textfields are empty.")
13l	[B52,500,400,50,50,4000,50,50, 1000]	Method call: createAircraft.
13m	[B52,500,400,50,50,4000,50,50, 1000]	warning("Aircraft name already in use.")
14a	[""]	warning("Unknown aircraft")
14b	[B52]	Method call: removeAircraft
15a	[""]	warning("Unknown aircraft.")
15b	[B52]	TTEditAircraftPanel[Replace] is shown.
16a	[B52,0,0,0,0,0,0,0,0]	All textfields is empty except the aircraft type textfield.
16b	[B52,500,400,150,30,10000,30,30, 5000]	Method call: replaceAircraft.
17a	["",B52,0]	warning("Invalid flight number.")
17b	[SK123,"",0]	warning("Unknown aircraft.")
17c	[SK123,B52,xxx]	warning("Invalid number of intermediate landings")
17d	[SK123,B52,-1]	warning("Number of intermediate landings must be at least zero")
17e	[SK123,B52,0]	All textfields are empty.
17f	[SK123,B52,0]	TTEditFlightPanel[Define Airport(1 of 2)] is shown.
18a	["",10,10,2000,8,00]	warning("Unknown airport.")
18b	[Airport1 (AP1),xx,10,2000,8,00]	warning("Invalid departure.")
18c	[Airport1 (AP1),40,10,2000,8,00]	warning("Invalid departure.")
18d	[Airport1 (AP1),10,10,2000,8,00]	All textfields are empty.

18e	[Airport1 (AP1),10,10,2000,8,00]	TTEditFlightPanel[Define Airport(2 of 2)] is shown.
19a	[Airport2 (AP2)]	Method call: insertFlight.
20a	[""]	warning("Unknown flight.")
20b	[SK123]	Method call: removeFlight.
21a	["",Airport1 (AP1),10,10,2000,12,00]	warning("Unknown flight.")
21b	[SK123,Airport1 (AP1),10,10,xxxx,12,00]	warning("Invalid departure year.")
21c	[SK123,Airport1 (AP1),10,10,2000,12,00]	All textfields are empty.
21d	[SK123,Airport1 (AP1),10,10,2000,12,00]	Method call: insertLanding.
22a	[SK123,Airport1 (AP1),Tue Oct 10 12:00 2000]	Method call: removeLanding.
22b	[SK123,Airport1 (AP1),Tue Oct 10 10:00 2000]	warning("Can't remove the destination or departure airport.")
23a	[xx]	warning("Invalid delay time.")
23b	[40]	Method call: setMinTimeSeparation.

B.2.2 GetAppropiatePanel Method Test

The getAppropiatePanel method in the ats.gui.EditorFrame is tested separately. This test is also created at runtime. By using the EditMenu in the EditorFrame we can test each case in the method separately. This is possible because we have created small demo sequences that will force the program to execute a specific branch. In each case in the method there are two branches (either beforeEvent is true or false). The following "a" tests are when the beforeEvent is true, and "b" tests are when beforeEvent is false. The "Current DemoEvent" column in the tabel indicates that DemoEvent there is at the current position in the demo sequence. This is stated because the getAppropiatePanel method gets a panel that suits the current DemoEvent and the state of the beforeEvent (controlled by the previous and next functions from the EditMenu).

Case	Current Demo-Event	Test	Expected Panel
1a	insertAirport	Previous	ASEditAirportPanel[Insert] (Old Values OK)
1b	insertAirport	Next	ASEditAirportPanel[Insert] (Initialized OK)
2a	removeAirport	Previous	ASEditAirportPanel[Remove] (Old Values OK)
2b	removeAirport	Next	ASEditAirportPanel[Remove] (Initialized OK)
3a	modifyAirport	Previous	ASEditAirportPanel[Modify] (Old Values OK)
3b	modifyAirport	Next	ASEditAirportPanel[Replace] (Values OK)
4a	replaceAirport	Previous	ASEditAirportPanel[Replace] (Old Values OK)
4b	replaceAirport	Next	AirspaceCanvas (OK)
5a	insertAirway1	Previous	ASEditAirwayPanel[Insert1] (Old Values OK)
5b	insertAirway1	Next	ASEditAirwayPanel[Insert2] (Initialized OK)
6a	insertAirway2	Previous	ASEditAirwayPanel[Insert2] (Old Values OK)
6b	insertAirway2	Next	ASEditAirwayPanel[Insert3] (Initialized OK)
7a	insertAirway3	Previous	ASEditAirwayPanel[Insert3] (Old Values OK)
7b	insertAirway3	Next	AirspaceCanvas (OK)
8a	removeAirway	Previous	ASEditAirwayPanel[Remove] (Old Values OK)
8b	removeAirway	Next	AirspaceCanvas (OK)
9a	insertFlight1	Previous	TTEditFlightPanel[Insert1] (Old Values OK)
9b	insertFlight1	Next	TTEditFlightPanel[Insert2] (Initialized OK)
10a	insertFlight2	Previous	TTEditFlightPanel[Insert2] (Old Values OK)
10b	insertFlight2	Next	TTEditFlightPanel[Insert3] (Initialized OK)
11a	insertFlight3	Previous	TTEditFlightPanel[Insert3] (Old Values OK)
11b	insertFlight3	Next	TTVisPanel(TimeTableApplet) (OK)
12a	removeFlight	Previous	TTEditFlightPanel[Remove] (Old Values OK)
12b	removeFlight	Next	TTVisPanel[TimeTableApplet] (OK)
13a	insertAircraft	Previous	TTEditAircraftPanel[Insert] (Old Values OK)
13b	insertAircraft	Next	TTVisPanel[TimeTableApplet] (OK)
14a	removeAircraft	Previous	TTEditAircraftPanel[Remove] (Old Values OK)
14b	removeAircraft	Next	TTVisPanel[TimeTableApplet] (OK)
15a	modifyAircraft	Previous	TTEditAircraftPanel[Modify] (Old Values OK)
15b	modifyAircraft	Next	TTEditAircrafPanel[Replace] (Values OK)
16a	replaceAircraft	Previous	TTEditAircraftPanel[Replace] (Old Values OK)
16b	replaceAircraft	Next	TTVisPanel[TimeTableApplet] (OK)
17a	insertLanding	Previous	TTEditLandingPanel[Insert] (Old Values OK)
17b	insertLanding	Next	TTVisPanel[TimeTableApplet] (OK)
18a	removeLanding	Previous	TTEditLandingPanel[Remove] (Old Values OK)
18b	removeLanding	Next	TTVisPanel[TimeTableApplet] (OK)
19a	setMinTimeSep.	Previous	PropertiesPanel[TimeTable] (Old Values OK)
19b	setMinTimeSep.	Next	DefaultPanel (OK)

Appendix C

DemoEvent Table

DemoEventID	Event description
1	insertAirport
2	removeAirport
3	modifyAirport
4	replaceAirport
5	insertAirway1 (airports selected)
6	insertAirway2 (segments defined)
7	insertAirway3 (method called)
8	removeAirway
21	insertFlight1 (info defined)
22	insertFlight2 (landings defined)
23	insertFlight3 (method called)
24	removeFlight
25	createAircraft
26	removeAircraft
27	modifyAircraft
28	replaceAircraft
29	insertLanding
30	removeLanding
31	setTime in time table
32	changeAirport in time table
33	flighteditor in time table
34	setMinTimeSeparation

Table C.1: List of EventID's matching their corresponding type of event.

Bibliography

[1] Arnold Field, *International Air Traffic Control – Management of the world's airspace*. Pergamon Press 1985.

[2] Günter Hake, *Kartographie I*. Walter de Gruyter 1975.

[3] Alliot et al, *CATS: A Complete Air Traffic Simulator*. 1997.

[4] Andreatta et al, *The flow management problem: recent algorithms*. 1997.

[5] Inselberg et al, *Multidimensional Lines II: Proximity and applications*. 1994.